Dissertation

Analysis and Design of Symmetric Cryptographic Algorithms

Submitted to the Faculty of Computer Science and Mathematics
of the University of Passau in Partial Fulfilment of the Requirements
for the Degree Doctor Rerum Naturalium

Philipp Jovanovic

May 2015

Thesis Advisors:	**Prof. Dr. Martin Kreuzer**	Chair of Symbolic Computation, Faculty of Computer Science and Mathematics, University of Passau
	Prof. Dr. Ilia Polian	Chair of Computer Engineering, Faculty of Computer Science and Mathematics, University of Passau
External Referee:	**Prof. Dr. Andrey Bogdanov**	Section for Cryptology, Department of Applied Mathematics and Computer Science, Technical University of Denmark

Abstract

This doctoral thesis is dedicated to the analysis and the design of symmetric cryptographic algorithms.

In the first part of the dissertation, we deal with *fault-based attacks* on cryptographic circuits which belong to the field of active implementation attacks and aim to retrieve secret keys stored on such chips. Our main focus lies on the cryptanalytic aspects of those attacks. In particular, we target block ciphers with a lightweight and (often) non-bijective key schedule where the derived subkeys are (almost) independent from each other. An attacker who is able to reconstruct one of the subkeys is thus not necessarily able to directly retrieve other subkeys or even the secret master key by simply reversing the key schedule. We introduce a framework based on differential fault analysis that allows to attack block ciphers with an arbitrary number of independent subkeys and which rely on a substitution-permutation network. These methods are then applied to the lightweight block ciphers LED and PRINCE and we show in both cases how to recover the secret master key requiring only a small number of fault injections. Moreover, we investigate approaches that utilize algebraic instead of differential techniques for the fault analysis and discuss advantages and drawbacks. At the end of the first part of the dissertation, we explore fault-based attacks on the block cipher Bel-T which also has a lightweight key schedule but is not based on a substitution-permutation network but instead on the so-called Lai-Massey scheme. The framework mentioned above is thus not usable against Bel-T. Nevertheless, we also present techniques for the case of Bel-T that enable full recovery of the secret key in a very efficient way using differential fault analysis.

In the second part of the thesis, we focus on *authenticated encryption* schemes. While regular ciphers only protect privacy of processed data, authenticated encryption schemes also secure its authenticity and integrity. Many of these ciphers are additionally able to protect authenticity and integrity of so-called associated data. This type of data is transmitted unencrypted but nevertheless must be protected from being tampered with during transmission. Authenticated encryption is nowadays the standard technique to protect in-transit data. However, most of the currently deployed schemes have deficits and there are many leverage points for improvements. With NORX we introduce a novel authenticated encryption scheme supporting associated data. This algorithm was designed with high security, efficiency in both hardware and software, simplicity, and robustness against side-channel attacks in mind. Next to its specification, we present special features,

security goals, implementation details, extensive performance measurements and discuss advantages over currently deployed standards. Finally, we describe our preliminary security analysis where we investigate differential and rotational properties of NORX. Noteworthy are in particular the newly developed techniques for differential cryptanalysis of NORX which exploit the power of SAT- and SMT-solvers and have the potential to be easily adaptable to other encryption schemes as well.

Zusammenfassung

Diese Doktorarbeit beschäftigt sich mit der Analyse und dem Entwurf von symmetrischen kryptographischen Algorithmen.

Im ersten Teil der Dissertation befassen wir uns mit *fehlerbasierten Angriffen* auf kryptographische Schaltungen, welche dem Gebiet der aktiven Seitenkanalangriffe zugeordnet werden und auf die Rekonstruktion geheimer Schlüssel abzielen, die auf diesen Chips gespeichert sind. Unser Hauptaugenmerk liegt dabei auf den kryptoanalytischen Aspekten dieser Angriffe. Insbesondere beschäftigen wir uns dabei mit Blockchiffren, die leichtgewichtige und eine (oft) nicht-bijektive Schlüsselexpansion besitzen, bei denen die erzeugten Teilschlüssel voneinander (nahezu) unabhängig sind. Ein Angreifer, dem es gelingt einen Teilschlüssel zu rekonstruieren, ist dadurch nicht in der Lage direkt weitere Teilschlüssel oder sogar den Hauptschlüssel abzuleiten indem er einfach die Schlüsselexpansion umkehrt. Wir stellen Techniken basierend auf differenzieller Fehleranalyse vor, die es ermöglichen Blockchiffren zu analysieren, welche eine beliebige Anzahl unabhängiger Teilschlüssel einsetzen und auf Substitutions-Permutations Netzwerken basieren. Diese Methoden werden im Anschluss auf die leichtgewichtigen Blockchiffren LED und PRINCE angewandt und wir zeigen in beiden Fällen wie der komplette geheime Schlüssel mit einigen wenigen Fehlerinjektionen rekonstruiert werden kann. Darüber hinaus untersuchen wir Methoden, die algebraische statt differenzielle Techniken der Fehleranalyse einsetzen und diskutieren deren Vor- und Nachteile. Am Ende des ersten Teils der Dissertation befassen wir uns mit fehlerbasierten Angriffen auf die Blockchiffre Bel-T, welche ebenfalls eine leichtgewichtige Schlüsselexpansion besitzt jedoch nicht auf einem Substitutions-Permutations Netzwerk sondern auf dem sogenannten Lai-Massey Schema basiert. Die oben genannten Techniken können daher bei Bel-T nicht angewandt werden. Nichtsdestotrotz werden wir auch für den Fall von Bel-T Verfahren vorstellen, die in der Lage sind den vollständigen geheimen Schlüssel sehr effizient mit Hilfe von differenzieller Fehleranalyse zu rekonstruieren.

Im zweiten Teil der Doktorarbeit beschäftigen wir uns mit *authentifizierenden Verschlüsselungsverfahren*. Während gewöhnliche Chiffren nur die Vertraulichkeit der verarbeiteten Daten sicherstellen, gewährleisten authentifizierende Verschlüsselungsverfahren auch deren Authentizität und Integrität. Viele dieser Chiffren sind darüber hinaus in der Lage auch die Authentizität und Integrität von sogenannten assoziierten Daten zu gewährleisten. Daten dieses Typs werden in nicht-verschlüsselter Form übertragen, müssen aber dennoch

gegen unbefugte Veränderungen auf dem Transportweg geschützt sein. Authentifizierende Verschlüsselungsverfahren bilden heutzutage die Standardtechnologie um Daten während der Übertragung zu beschützen. Aktuell eingesetzte Verfahren weisen jedoch oftmals Defizite auf und es existieren vielfältige Ansatzpunkte für Verbesserungen. Mit NORX stellen wir ein neuartiges authentifizierendes Verschlüsselungsverfahren vor, welches assoziierte Daten unterstützt. Dieser Algorithmus wurde vor allem im Hinblick auf Einsatzgebiete mit hohen Sicherheitsanforderungen, Effizienz in Hardware und Software, Einfachheit, und Robustheit gegenüber Seitenkanalangriffen entwickelt. Neben der Spezifikation präsentieren wir besondere Eigenschaften, angestrebte Sicherheitsziele, Details zur Implementierung, umfassende Performanz-Messungen und diskutieren Vorteile gegenüber aktuellen Standards. Schließlich stellen wir Ergebnisse unserer vorläufigen Sicherheitsanalyse vor, bei der wir uns vor allem auf differenzielle Merkmale und Rotationseigenschaften von NORX konzentrieren. Erwähnenswert sind dabei vor allem die für die differenzielle Kryptoanalyse von NORX entwickelten Techniken, die auf die Effizienz von SAT- und SMT-Solvern zurückgreifen und das Potential besitzen relativ einfach auch auf andere Verschlüsselungsverfahren übertragen werden zu können.

Acknowledgements

After four years of working on my PhD and after months of thesis writing, it is time to come to the most important part of this dissertation. I am indebted to many people who I have met and worked with and who have supported me along the way. I doubt that my path in life would have turned out the way it did without their involvement. It is therefore a real pleasure for me to hereby take the opportunity and express my acknowledgements to all of them.

First of all, I would like to thank my supervisors, Martin Kreuzer and Ilia Polian, for accepting me as a PhD student, for continuously supporting me in all matters, be it scientific or otherwise, and especially for their open-door policy. Moreover, I would like to thank them for allowing me to pursue my own path in research which was in many cases only loosely connected to the main research topics of their respective groups. I also want to express my deep appreciation to Andrey Bogdanov who kindly agreed to serve as the external referee for this dissertation.

During my time as a PhD student at the University of Passau, I had the fortune to meet and collaborate with many very smart and highly knowledgeable researchers and I want to acknowledge in particular my co-authors: Jean-Philippe Aumasson, Christof Beierle, Wayne Burleson, Martin Kreuzer, Raghavan Kumar, Martin Lauridsen, Gregor Leander, Atul Luykx, Bart Mennink, Samuel Neves, Ilia Polian, and Christian Rechberger.

My special thanks go to Jean-Philippe Aumasson and Samuel Neves for teaming up with me back in early 2013 to participate in CAESAR, and for the countless, evening-filling, scientific and non-scientific discussions on IRC and during the all too rare occasions when we met in person. Working with you was lots of fun and I learned so much in the process. Thanks a lot, you are the best.

Furthermore, I would like to thank Christian Rechberger for enabling me to visit DTU in 2014 and for allowing me to work with him and his team. Many thanks also to all the other members of DTU's Cryptology Research Group, and in particular to Andrey Bogdanov, Stefan Kölbl, Martin Lauridsen, Arnab Roy, Tyge Tiessen, and Elmar Tischhauser for making my stay in Copenhagen such a great and memorable experience.

Moreover, I want to express my acknowledgements to all my current and former colleagues and friends at the University of Passau. Particularly, I would like to thank Markus Kriegl, Jan Limbeck, Severin Neumann, Stefan Schuster, and Thomas Stadler for the countless both scientific and non-scientific discussions and their readiness to help in any situation. I am also deeply grateful to our secretaries Nathalie Vollstädt and Anna

Weikelsdorfer for their constant support in all organisational concerns. Thanks, you all contributed to this thesis in one way or the other.

My deepest and sincerest gratitude goes also to my family, my parents Christine and Michael, and my brother David. Thank you so much for your never-ending support, for the possibilities you have given me in life, and for always being there when needed. Moreover, my thanks go as well to my family (in-law), Ingrid, Alois, and Dominic, for all the enjoyable moments that I was allowed to share with you over the past years.

Finally, I would like to express my utmost gratefulness to Sabrina, my wonderful girlfriend. You have accompanied, encouraged, and supported me with your love, friendship, understanding, and humour through all the ups and downs of my PhD and beyond. Without you, I would have never made it this far. Thank you for everything.

Philipp Jovanovic
Passau, May 2015

Contents

Acknowledgements i

Motivation vii

List of Symbols xi

1 Introduction 1
- 1.1 Cryptography . . . 1
  - 1.1.1 Block Ciphers . . . 2
  - 1.1.2 Stream Ciphers . . . 6
  - 1.1.3 Hash Functions . . . 7
  - 1.1.4 Message Authentication Codes . . . 8
  - 1.1.5 Authenticated Encryption Schemes . . . 8
- 1.2 Cryptanalysis . . . 17
  - 1.2.1 Brute-Force Attacks . . . 20
  - 1.2.2 Differential Attacks . . . 23
  - 1.2.3 Linear Attacks . . . 30
  - 1.2.4 Algebraic Attacks . . . 32
  - 1.2.5 Rotational Attacks . . . 35
  - 1.2.6 Implementation Attacks . . . 35
- 1.3 Security Notions . . . 40

2 Fault-based Attacks on the Block Ciphers LED and PRINCE 43
- 2.1 Introduction . . . 43
- 2.2 The Block Cipher LED . . . 44
  - 2.2.1 General Layout . . . 44
  - 2.2.2 Round Function . . . 46
- 2.3 The Block Cipher PRINCE . . . 48
  - 2.3.1 General Layout . . . 48
  - 2.3.2 Round Function . . . 48
- 2.4 Fault Attacks on LED-64 . . . 50
  - 2.4.1 Fault Models . . . 50
  - 2.4.2 Fault Equations . . . 51

2.4.3 Key Filtering Mechanisms . . . 54
2.4.4 Experimental Results . . . 60
2.4.5 Extensions of the Fault Attack . . . 61
2.5 Multi-Stage Fault Attacks on LED-128 and PRINCE . . . 64
2.5.1 The Multi-Stage Fault Attack Framework . . . 64
2.5.2 Applications to LED-128 . . . 66
2.5.3 Experimental Results . . . 68
2.5.4 Applications to PRINCE . . . 69
2.5.5 Experimental Results . . . 72
2.5.6 Extensions of the Fault Attacks . . . 73
2.6 Algebraic Fault Attacks on LED-64 . . . 74
2.6.1 Algebraic Representation of LED . . . 74
2.6.2 Algebraic Representation of the LED Fault Equations . . . 79
2.6.3 Experimental Results . . . 79
2.7 Conclusion . . . 80

3 Fault-based Attacks on the Bel-T Block Cipher Family 83
3.1 Introduction . . . 83
3.2 The Block Cipher Bel-T . . . 84
3.3 Fault Attacks on Bel-T . . . 85
3.3.1 Bel-T-128 . . . 87
3.3.2 Bel-T-192 . . . 89
3.3.3 Bel-T-256 . . . 90
3.3.4 Experimental Results . . . 90
3.4 Practical Issues and Countermeasures . . . 91
3.5 Conclusion . . . 92

4 NORX: Parallel and Scalable Authenticated Encryption 95
4.1 Introduction . . . 95
4.2 Specification . . . 99
4.2.1 Preliminaries . . . 99
4.2.2 Parameters and Interface . . . 100
4.2.3 Layout Overview . . . 101
4.2.4 The Round Function . . . 102
4.2.5 Encryption Mode . . . 103
4.2.6 Decryption Mode . . . 111
4.2.7 Datagrams . . . 113
4.3 Security Goals . . . 114
4.4 Features . . . 119
4.4.1 List of Characteristics . . . 119
4.4.2 Recommended Parameter Sets . . . 121

4.4.3 Performance . . . 121
4.5 Design Rationale . . . 130
4.5.1 The Parallel MonkeyDuplex Construction . . . 130
4.5.2 The Functions F, G, and H . . . 131
4.5.3 Selection of Constants . . . 133
4.5.4 Number of Rounds . . . 135
4.5.5 The Padding Rule . . . 137
4.6 Conclusion . . . 137

5 Analysis of NORX **139**
5.1 Introduction . . . 139
5.2 General Observations on G and F . . . 139
5.2.1 Fix Points . . . 139
5.2.2 Weak States . . . 140
5.2.3 Algebraic Properties . . . 141
5.2.4 Slide Attacks . . . 142
5.3 Differential Cryptanalysis . . . 142
5.3.1 Simple Differentials . . . 143
5.3.2 Impossible Differentials . . . 147
5.3.3 NODE – NORX Differential Search Engine . . . 149
5.4 Rotational Cryptanalysis . . . 163
5.5 Conclusion . . . 165

Bibliography **167**

Test Vectors for NORX **187**

Publications **195**

Motivation

Cryptology consists of two interacting counterparts: *cryptography*, the science of designing secure communication channels in presence of third parties, on the one hand, and *cryptanalysis*, the science of evaluating the security of cryptographic constructions, on the other. Traditionally deployed by military and secret services — we refer to Kahn [146] for an extensive treatment of the early history and to the seminal publications of Kerckhoffs [151] and Shannon [223, 224] for the foundations of modern cryptology — the situation changed drastically in the past decades where cryptology found its way into our everyday life. The main reason, undoubtedly, is the *Digital Revolution* triggered by the evolution of the computer and the introduction of the Internet, which led to a rapid-increasing influence of technology and digital media on basically every aspect of modern society.

Protection of sensitive digital data against unauthorised access is nowadays not only a highly relevant topic for industry and governments but a concern for principally everyone. Security of data in phone calls, email, mobile messaging, online shopping, online banking or in emerging fields such as electronic currency systems (e.g. bitcoin [193]), smart grids, or the Internet-of-Things would be unthinkable without sound cryptographic constructions. The strength of the cryptographic protection is determined by the (in)feasibility of deriving secret information by unauthorized parties. Goals of modern cryptology include for example confidentiality, integrity, authenticity, anonymity, and non-repudiation to name just a few of the many objectives.

Although there exist many cryptographic schemes which are considered secure, there is no single, universally applicable solution. Due to ever-changing requirements and new application fields, there is a constant demand for innovative solutions that master emerging challenges. Another problem is that deprecated cryptographic constructions are still relatively widespread. Those designs often date back to times when basically no one could anticipate the dimensions current technologies, such as today's Internet, could reach and are now usually outdated and not suitable for usage in modern applications. One prominent example is the RC4 stream cipher, which was designed by Rivest in 1987, became publicly known in 1994, and found widespread adoption due to its simplicity and relatively good performance. The security of RC4 has been analysed very thoroughly over the past decades and many weaknesses in the algorithm itself and in systems it has been deployed in were uncovered [5, 114, 157, 221, 232]. Cryptographers have been advising against its usage already for years but phasing out such a widely deployed system is usually a difficult task and a very slow process due to complex interdependencies, issues

with backwards compatibility, and various other reasons. For example, in early 2014, RC4 was still one of the most widely used ciphers in implementations of the *Transport Layer Security* (TLS) protocol which secures communication on the Internet. In 2015, the *Internet Engineering Task Force* (IETF) finally prohibited its usage in TLS [207]. There are many similar examples of cryptographic primitives whose weaknesses have been revealed thanks to continuous advances in cryptanalysis. Understandably, the interest is huge to replace legacy designs by new, modern variants that amend the flaws of their older counterparts, provide new features, and often promise a drastic reduction of operational costs. Moreover, modern ciphers are commonly designed with big enough security margins so that they are able to resist future cryptanalytic or computational breakthroughs. This is absolutely vital to ensure security in the long run since many cryptographic primitives are used for decades, as can be seen on the example of RC4.

One big threat to many of the known cryptographic systems are quantum computers [197]. While there currently exist early prototypes that can only be used for very elementary computations, the interest from academia, industry, and governments alike is substantial to construct a real and practically usable quantum computer. It would provide huge computational benefits in comparison to classical computers. Although there are still many challenges to overcome, unforeseen innovations in engineering could quickly lead to the construction of a first quantum computer with a reasonable number of quantum-bits (qubits). We refer to the quantum algorithms of Grover [120] and Shor [226] which yield, in comparison to the best known classical algorithms, considerable speed-ups to the problems of database search and integer factorisation. Ciphers such as RSA which belongs to one of the most widely deployed public-key crypto systems and whose security is based on the hardness of integer factorisation could be broken easily by a quantum computer equipped with enough qubits. It is therefore no surprise that post-quantum cryptography is a highly active research field where cryptographers investigate new systems that remain secure even in presence of quantum computers.

On the other end of the spectrum there is a huge interest in the field of lightweight cryptography motivated by pervasive computing, enabled through small mobile and embedded devices, like RFID chips and nodes of sensor networks. These appliances increasingly find their way into our everyday life and are often utilised to process sensitive (personal) data, for example in the form of financial or medical information. Obviously, protecting such information is essential and is in large parts achieved through the deployment of cryptographic methods. However, the acceptable complexity of cryptographic algorithms implementable on low-end devices is typically restricted by stringent cost constraints, by power consumption limits due to battery life-time, or by heat dissipation issues. The design of cryptographic primitives that provide acceptable security against conventional cryptanalysis and implementation attacks, and that can be realised on devices with strictly limited resources is a very challenging task and has raised significant interest in the last few years [158]. It is therefore no surprise that numerous new algorithms [15, 61, 71, 72, 73, 78, 85, 86, 125, 126] were proposed addressing the manifold challenges of lightweight

cryptography.

To summarize, cryptology is a highly active and challenging research field with countless unsolved and practice-oriented issues. The ever increasing necessity and demand for security and privacy of digital communication of our high-tech society in the information age ensures that research in cryptology will stay relevant for many years to come.

Research Contributions and Outline

This thesis deals with research problems in symmetric cryptology, where it is assumed that the communicating parties share a secret key. In particular, we investigate techniques for fault-based cryptanalysis of block ciphers, discuss the design of a novel authenticated encryption scheme, and also describe our security evaluation of the latter. The outline of the thesis is as follows.

In Chapter 1, we discuss basic concepts from symmetric cryptology. We introduce block ciphers, stream ciphers, hash functions, and message authentication codes, the basic primitives from symmetric cryptography, and also discuss authenticated encryption schemes, a more advanced construction. Moreover, we give an introduction to the basic tools of cryptanalysis including brute-force, differential, linear, algebraic, rotational, and implementation attacks. The purpose of this chapter is to initiate basic terminology required later on in the thesis.

In Chapter 2, we discuss techniques for fault analysis of the lightweight block ciphers LED and PRINCE. We start with a fault-based attack on LED-64 and introduce filtering techniques which quickly eliminate wrong key hypotheses. We show that the number of remaining key candidates is already small enough after a single fault injection to make exhaustive search feasible. We also motivate why those techniques are not directly applicable to LED-128 and PRINCE. Afterwards, we present a generalisation of the LED-64 attack which leads to the multi-stage fault attack framework and allows differential fault analysis of both LED-128 and PRINCE. We show that in both cases between 3 and 5 fault injections are sufficient for a successful reconstruction of the entire 128-bit key and also present the results from our extensive simulation-based experiments. Finally, we discuss an extension of the LED-64 attack to an algebraic setting. The results of this chapter are published (partially as preprints) in [138, 139, 140]. Furthermore, in [172, 173], the applicability of the fault analysis techniques in combination with new methods for high-precision fault injections is investigated.

In Chapter 3, we present differential fault analysis of the block cipher family Bel-T which has been adopted recently as a national standard of the Republic of Belarus. Our attacks successfully recover the secret key of the 128-bit, 192-bit, and 256-bit versions of Bel-T using 4, 7, and 10 fault injections, respectively. We also discuss the feasibility of the required fault injections and show the results from our comprehensive simulation-based experiments. The results of this chapter are published in [143].

In Chapter 4, we introduce NORX, a novel authenticated encryption scheme with support

for associated data, which was submitted in 2014 as a first-round candidate to CAESAR, the *Competition for Authenticated Encryption: Security, Applicability and Robustness*. NORX was designed with a focus on high-security, simplicity, high-performance, and side-channel robustness. It is based on the monkeyDuplex construction which belongs to the family of sponge functions and features an original domain separation scheme for simple processing of header, payload, and trailer data. NORX was optimised for efficiency in both soft- and hardware, having a core suitable for vectorized implementations, almost byte-aligned rotations, no secret-dependent memory lookups, and only bitwise logical operations. On a Haswell processor, a serial version of NORX runs at 2.51 cycles per byte. Simulations of a hardware architecture for 180 nm UMC ASIC give a throughput of approximately 10 Gbps at 125 MHz. The main results of this chapter are published in [20] and further improvements on the generic security bounds can be found in [141]. In addition, a talk about CAESAR and NORX was given at the 31st Chaos Communication Congress (31C3) [17].

In Chapter 5, we present a thorough security analysis of NORX and focus, in particular, on differential and rotational properties. After the discussion of some basic properties, we introduce mathematical models that describe differential propagation with respect to the non-linear operation of NORX. Afterwards, we present NODE, the NORX differential search engine, which is an adaptation of a framework previously proposed for ARX designs, allowing us to automate the search for differentials and characteristics. We give upper bounds on the differential probability for a small number of steps of the NORX core permutation. For example, in a scenario where an attacker can only modify the nonce during initialisation, we show that there are no differential characteristics with higher probabilities than 2^{-67} (32-bit) and 2^{-62} (64-bit) after only one round. Furthermore, we describe how we found the best characteristics for four rounds, which have probabilities of 2^{-584} (32-bit) and 2^{-836} (64-bit), respectively. Finally, we discuss some rotational properties of the core permutation which yield some first, rough security bounds and can be used as a basis for future studies. The results of this chapter are published in [19].

List of Symbols

$\mathbb{N}$	set of natural numbers including 0
$\mathbb{Z}$	ring of integers
$\mathbb{Z}_n$	residue class ring of integers modulo n
$K[x_1, \ldots, x_n]$	polynomial ring in indeterminates $x_1, \ldots, x_n$ over the field K
$\mathbb{Q}$	field of rational numbers
$\mathbb{F}_{p^n}$	finite field with p^n elements, p prime, $n \geq 1$
$\mathbb{F}_2^n$	$\mathbb{F}_2$-vector space of bit strings $X = (x_0, \ldots, x_{n-1})$ with length $n \geq 1$
$\mathbb{F}_2^*$	set of bit strings with arbitrary but finite length
0^n	bit string consisting of n zeroes
$\lvert X \rvert$	length of bit string X in bits
$\lvert X \rvert_r$	length of bit string X in r-bit blocks
$\mathsf{hw}(X)$	Hamming weight of bit string X
$\lfloor X \rfloor_n$	truncation of bit string X to its first, i.e. least-significant, n bits
$X \parallel Y$	concatenation of bit strings X and Y
$X \ll n$	left-shift of bit string X by n bits
$X \gg n$	right-shift of bit string X by n bits
$X \lll n$	cyclic left-rotation of bit string X by n bits
$X \ggg n$	cyclic right-rotation of bit string X by n bits
$\neg$, $\wedge$, $\vee$, $\oplus$	bitwise logical `NOT`, `AND`, `OR`, and `XOR`
$\boxplus$, $\boxminus$	integer addition and subtraction
$a \leftarrow b$	assignment of value b to the variable a
$x \xleftarrow{\$} X$	sample x uniformly at random from the set X
$f \circ g$	composition of functions f and g

Chapter 1

Introduction

1.1 Cryptography

There are three major categories of cryptographic primitives, namely *unkeyed*, *symmetric*, and *asymmetric* algorithms. Figure 1 gives an overview on the most common cryptographic primitives. The distinguishing property of those categories is the different usage of key material: unkeyed algorithms do not require any secret information to be used. Symmetric algorithms use a single *secret key* that is shared among all valid communication partners and is used by all of them to execute cryptographic operations such as encryption and decryption of data. For the usage of asymmetric algorithms each participant is required to posses a pair of keys, a *public key* and a *private key*. The two keys of a participant are strongly related to each other and each has its own purpose which can be roughly summarised as follows: the public key is used for encryption or verification of digital signatures, whereas the private key is used for decryption or creation of digital signatures. In practice, the different kinds of primitives are usually not just used on their own but instead are combined to form *cryptographic protocols*. This thesis focusses on symmetric cryptography and the section at hand introduces its core principles. Additionally, we also give a brief overview on hash functions due to their important role in cryptography. For the other topics, we refer the interested reader to standard literature about cryptography [202].

There are many goals that can be achieved with (symmetric) cryptography, but three of the fundamental ones are:

- **Confidentiality.** It ensures that an adversary who has access to a communication channel is not able to derive information about the content of messages exchanged by the communications partners.
- **Integrity.** It ensures that an adversary who has access to a communication channel is not able to modify the content of exchanged messages in an unauthorised way. In other words, it prevents an active adversary from tampering with transmitted messages without the manipulation being noticed.
- **Authenticity.** It ensures that an adversary who has access to a communication channel is not able to modify the information about the origin of exchanged messages,

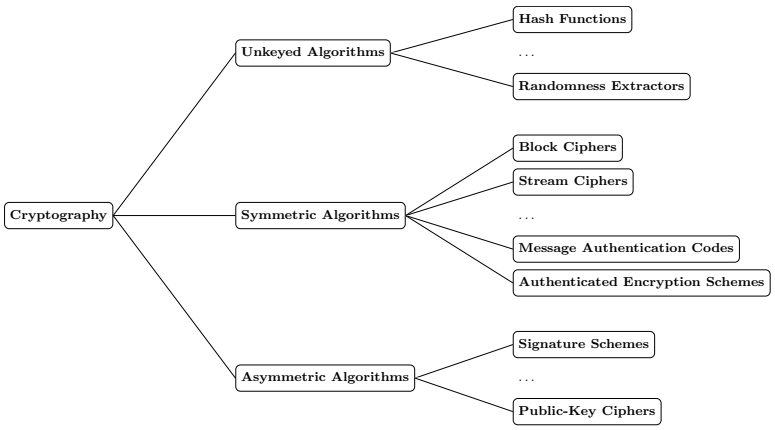


Figure 1: Categories of common cryptographic algorithms.

i.e. it prevents an attacker from impersonating as a valid source of messages to any of the true communication partners.

Different kinds of symmetric cryptographic constructions can be specified which achieve a varying number of the above goals. In the following, we introduce the basic symmetric cryptographic primitives, as listed in Figure 1, and describe their respective roles in achieving the three goals above. As a basis, we use standard literature on (symmetric) cryptography such as [162, 202].

1.1.1 Block Ciphers

Block ciphers are a core building block of symmetric cryptography and ensure the confidentiality of processed data. They are often used to design other cryptographic primitives, such as stream ciphers, hash functions or message authentication codes. In the following, we introduce the basic definition and discuss thereafter common approaches for the construction of block ciphers.

Let $k, b \geq 1$. A *block cipher* is a tuple $\Pi = (\mathcal{E}, \mathcal{D})$ such that the *encryption function*

$$\mathcal{E} : \mathbb{F}_2^k \times \mathbb{F}_2^b \to \mathbb{F}_2^b,\ (K, M) \mapsto C$$

is a permutation on the set of *plaintexts* $M \in \mathbb{F}_2^b$ for a fixed *secret key* $K \in \mathbb{F}_2^k$. The value b is also called the *block size*. The inverse of the encryption function $\mathcal{E}^{-1}$, also called the *decryption function*, is denoted by $\mathcal{D}$. In particular, the equation $\mathcal{D}_K(\mathcal{E}_K(M)) = M$ holds for all plaintexts $M \in \mathbb{F}_2^b$ and a fixed secret key $K \in \mathbb{F}_2^k$, where we denote $\mathcal{E}_K(\cdot) = \mathcal{E}(K, \cdot)$ and $\mathcal{D}_K(\cdot) = \mathcal{D}(K, \cdot)$, respectively.

Common values for k are 64, 80, 96, 128, 192, and 256 bits and for b often values of 64, 128 or 256 bits are used. Block ciphers specify families of permutations. The block

size of b bits determines the space of all possible permutations, while the key size of k bits determines the number of permutations that are actually created. More precisely, for a given key size of k bits there exist 2^k different keys, and choosing one of them (at random) selects one of the permutations on the set of 2^b inputs (at random). There are $(2^b)!$ different permutations on b-bit input blocks which corresponds roughly to the value $2^{(b-1)2^b}$ by Stirling's approximation. Usually, one also demands that keys which are related to each other in some way, yield permutations sharing no recognisable relations, which could be exploited in cryptanalytic attacks otherwise.

Block ciphers are commonly constructed in an iterative way based on bijective, key-dependent *round functions* $f_i(K_i, \cdot)$ which operate on b-bit blocks of data. Note that K_i denotes the ith *round key* for $i \in \{0, \ldots, r-1\}$ and r denotes the *number of rounds*. Thus, the encryption function of such an iterative block cipher can be described by

$$\mathcal{E}(K, \cdot) = f_{r-1}(K_{r-1}, \cdot) \circ f_{r-2}(K_{r-2}, \cdot) \circ \cdots \circ f_1(K_1, \cdot) \circ f_0(K_0, \cdot)$$

where $\circ$ denotes function composition. Analogously, decryption can be described by

$$\mathcal{D}(K, \cdot) = f_0^{-1}(K_0, \cdot) \circ f_1^{-1}(K_1, \cdot) \circ \cdots \circ f_{r-2}^{-1}(K_{r-2}, \cdot) \circ f_{r-1}^{-1}(K_{r-1}, \cdot)$$

where $f_i^{-1}(K_i, \cdot)$ denotes the inverse to $f_i(K_i, \cdot)$. To obtain the round keys K_i the *master key* K is expanded using a *key schedule* g, meaning

$$g : \mathbb{F}_2^k \to \mathbb{F}_2^{qr} : K \mapsto (K_0, K_1, \ldots, K_{r-2}, K_{r-1})$$

where q denotes the bit size of a round key. In many cases, q coincides with the block size b, i.e. $q = b$. Figure 2 illustrates the encryption function of such an iterative block cipher. Depending on the design of the block cipher often so-called *whitening keys* are used before and after the application of all the round functions to mask plain- and ciphertext, respectively.

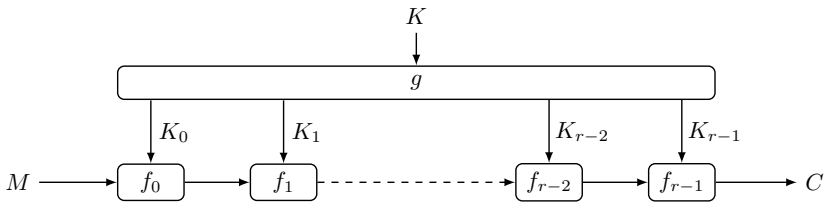


Figure 2: Encryption function of an iterative block cipher.

If the block cipher design can be modelled as a sequence of unkeyed round functions interleaved with addition of round keys using bitwise logical XOR, then we usually speak of a *key-alternating* [97] construction. Note that Feistel ciphers can also be key-alternating in some sense but cannot necessarily be modelled in such a way directly.

Now we give a brief overview on three common design approaches for block ciphers, namely *Feistel networks*, *substitution-permutation networks*, and *Lai-Massey schemes*. Figure 3 illustrates the concepts of the rounds functions for each of the aforementioned design strategies.

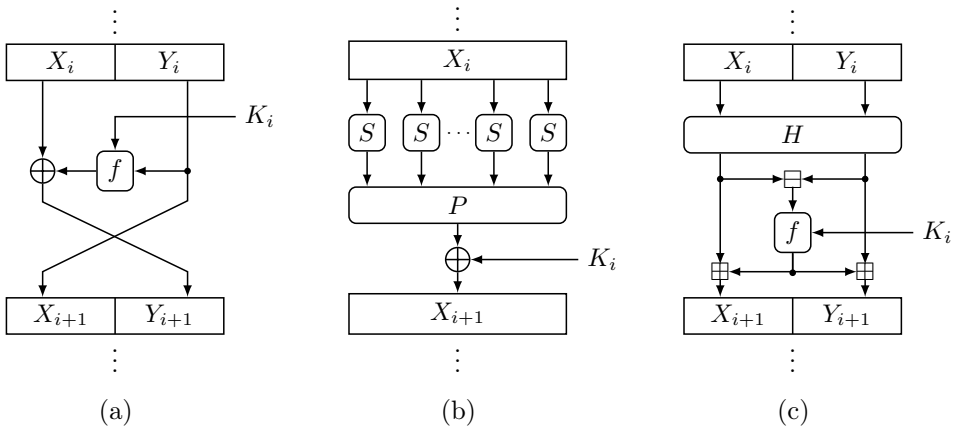


Figure 3: Round functions of (a) Feistel networks, (b) substitution-permutation networks, and (c) Lai-Massey schemes.

Feistel Networks

Block ciphers based on Feistel networks, see Figure 3a, have their state split into two halves, usually denoted by a left one X_i and a right one Y_i, for $0 \leq i \leq r$. The plaintext is loaded into $X_0 \parallel Y_0$. In a single round, a non-linear function f depending on a round key K_i is applied onto one of the halves and the result is XORed to the other. Finally, the two halves are swapped, which also finishes the round. Thus, a single encryption round of a Feistel network can be described through

$$\begin{aligned} X_{i+1} &= Y_i \\ Y_{i+1} &= X_i \oplus f_{K_i}(Y_i) \end{aligned}$$

which is also depicted in the first illustration of Figure 3. This process is repeated as long as specified by the number of rounds r. The ciphertext finally corresponds to $X_r \parallel Y_r$. Note that f does not necessarily have to be bijective. Decryption can be achieved in a very similar way to encryption, by simply exchanging the roles of X_i and Y_i and possibly adapting the key schedule. The similarity of encryption and decryption functions obviously helps to cut down costs, for example when the cipher is implemented in hardware. Therefore, it is not surprising that block ciphers based on Feistel networks

are often used in devices which only have access to very limited resources. Prominent representatives of this category are the Data Encryption Standard (DES), the AES finalist Twofish [219], or SIMON [27], a lightweight block cipher designed and published by the NSA.

Substitution-Permutation Networks

Another prevalent approach to design block ciphers are substitution-permutation networks (SPN), see Figure 3b. The basic building blocks of SPN block ciphers are a substitution layer S, which transforms the state in a non-linear way through parallel substitution of groups of bits according to certain substitution tables, better known as *S-boxes*, a linear permutation layer P, which permutes either single bits or entire groups of bits, and finally addition of a round key K_i usually using bitwise `XOR` or integer addition. Sometimes the round function also includes an operation for addition of a *round constant*, to make the single rounds distinct from each other, which impedes certain kind of attacks such as *slide attacks* [65]. The basic variant of the round function can be described as follows:

$$X_{i+1} = P(S(X_i)) \oplus K_i \ .$$

SPN block ciphers are by definition key-alternating and the decryption function is usually quite different from encryption compared to their Feistel network based counterparts. Lately however, there have been increased efforts to create SPN ciphers using *involutive* building blocks which allow to specify encryption and decryption functions in similar ways. For instance, PRINCE [78] falls into the latter category. Other prominent examples of substitution-permutation network based block ciphers include AES [96], PRESENT [71], and LED [126]. In Chapter 2, we analyse the ciphers LED and PRINCE against certain cryptanalytic attacks.

Lai-Massey Schemes

A third but less common option for block cipher design is the so-called Lai-Massey scheme, see Figure 3c. Like Feistel networks, the scheme works with a state divided in two parts X_i and Y_i. The building blocks of the round function are a half-round function H and a keyed transformation f_{K_i}, where K_i denotes the round key. The function H commonly updates the left state element X_i by application of a special operation σ, i.e. $(\sigma(X_i), Y_i)$, which is required to prevent trivial distinguishing attacks [236]. The above components are then combined as follows:

$$\begin{aligned} (A_i, B_i) &= H(X_i, Y_i) \\ C_i &= f_{K_i}(A_i \boxminus B_i) \\ (X_{i+1}, Y_{i+1}) &= (A_i \boxplus C_i, A_i \boxplus C_i) \ . \end{aligned}$$

Analogously to Feistel block ciphers, the function f does not have to be invertible. The Lai-Massey scheme was introduced alongside of IDEA [176]. Other representatives are FOX [144], now better known as IDEA-NXT, and, to some extent, also Bel-T [98], the national encryption standard of the Republic of Belarus. We analyse Bel-T in more detail in Chapter 3.

Block Cipher Modes

Block ciphers can encrypt only a single fixed-size block of data at a time. To be able to process messages of arbitrary length, though, a block cipher has to be used together with a proper *block cipher mode of operation*. The first block cipher modes were proposed and standardised by NIST for usage with DES in *FIPS 81* [194] and were later also standardised for the usage with AES [195]. The basic modes include *Electronic Codebook* (ECB), *Block Cipher Chaining* (CBC), *Ciphertext-Feedback* (CFB), *Output-Feedback* (OFB), and *Counter* (CTR). We are not discussing the details of those modes at this point but instead refer the interested reader to standard literature [162, 202].

1.1.2 Stream Ciphers

Stream ciphers accompany block ciphers as the second important class of symmetric-key primitives. While block ciphers encrypt data block-wise, a stream cipher achieves encryption by first producing a pseudo-randomly generated stream of bits (sometimes in the form of whole bit-blocks), the *key stream*, of the same size as the message and by XORing this key stream subsequently to the plaintext to obtain the ciphertext. This property makes stream ciphers very flexible as usually no message-padding or special mode of operation is required and arbitrary-sized messages can be processed right-away. However, note that a given block cipher can be easily transformed into a stream cipher using, for example, the already above mentioned counter mode (CTR).

Let $k, n \geq 1$. A *stream cipher* $\mathcal{S}$ is specified by

$$\mathcal{S} : \mathbb{F}_2^k \times \mathbb{F}_2^n \times \mathbb{F}_2^* \to \mathbb{F}_2^*,\ (K, N, M) \mapsto S \oplus M$$

where K is a secret key, N is either a *initialisation vector* (IV) or *nonce*, M a message, and S a pseudo-randomly generated key stream of length $|M|$. The ciphertext C corresponds to the output $S \oplus M$ of $\mathcal{S}$. Since XOR is an involution, the same function can be used for decryption with exchanged roles of C and M. Hence, the plaintext can be recovered by simply computing $\mathcal{S}(K, N, C) = S \oplus C = M$.

Note that there is a difference between an IV and a nonce: IVs are required to be chosen uniformly at random while nonces only have to be unique in order to guarantee the security of the algorithm. Thus, a nonce can be implemented through a simple counter, which is not possible for an IV. Whether an IV or nonce has to be used depends on the concrete cryptographic construction.

Undoubtedly, one of the most well-known stream ciphers is RC4, which was invented by Rivest in 1987 and is still in wide use today in spite of the many discovered weaknesses that very often allow to mount practical attacks on RC4. Modern and secure counterparts include Salsa20 [41], ChaCha [40], Trivium [86] and Grain-128a [2].

1.1.3 Hash Functions

Cryptographic hash functions are another important primitive and can be used to ensure the *integrity* of processed data. In symmetric cryptography, they are also often used as building blocks for other cryptographic primitives such as stream ciphers, message authentication codes, or authenticated encryption schemes. Although not covered in-depth within the thesis at hand, we nevertheless discuss them briefly below due to their importance and for the sake of completeness. Hash functions do not use a secret key, unlike the symmetric primitives discussed so far, and compress an arbitrary-sized but finite input to a fixed-sized output. The latter can be seen as the fingerprint, i.e. the "unique" identifier of the input. These primitives belong to the family of so-called *one-way functions* which means that they are considered practically impossible to invert. Due to their versatility, hash functions are often referred to as the "Swiss-army knife" of cryptography. Their field of application includes, but is not restricted to, data integrity checks, password verification, pseudorandom number generation, and message authentication. The formal definition of a hash function is as follows.

Let $n \geq 1$. A (cryptographic) *hash function* is a mapping

$$\mathcal{H} : \mathbb{F}_2^* \to \mathbb{F}_2^n,\ M \mapsto H$$

that takes as input a message M of arbitrary but finite length and compresses it into a fixed-size *digest* or *hash* H of length n.

Informally, a cryptographic hash function should be indistinguishable from a random function with the same parameters and it should fulfil the following four properties:

- **Efficiency.** Given the input M it is easy to compute $\mathcal{H}(M)$.
- **Collision Resistance.** Finding two distinct inputs $M \neq M'$, such that $\mathcal{H}(M) = \mathcal{H}(M')$ should require at least $2^{n/2}$ operations.
- **Preimage Resistance.** Given an image Z of $\mathcal{H}$, it is hard to find an input M such that $\mathcal{H}(M) = Z$.
- **Second Preimage Resistance.** Given an input M, it is hard to find a second input M', such that $\mathcal{H}(M) = \mathcal{H}(M')$.

Due to the above properties the value n is required to have a certain size. Common choices for n are 160, 256 and 512 bits.

Well-known cryptographic hash functions include MD5, SHA1, SHA256, SHA512, KECCAK [47], which was the winner of the SHA3 competition [222] and is now the new SHA3 standard, Grøstl [118], BLAKE [16], and BLAKE2 [21].

1.1.4 Message Authentication Codes

Hash functions that take a secret key as an additional input, are better known as message authentication codes. These primitives do not only provide data integrity but also allow to verify the authenticity of a message. This means that the receiver of a message can verify that it originates from a valid sender, namely the one with whom the receiver had exchanged the secret key before.

Concretely, a *message authentication code* (MAC) is a tuple $(\mathcal{T}, \mathcal{V})$ consisting of a *tag generation function* $\mathcal{T}$ and a *tag verification function* $\mathcal{V}$. The tag generation function is specified by

$$\mathcal{T} : \mathbb{F}_2^k \times \mathbb{F}_2^* \to \mathbb{F}_2^t,\ (K, M) \mapsto T$$

and takes as input a secret key K and an arbitrary long message M and compresses it into a fixed-size *authentication tag* T of length t. The tag verification function is specified through

$$\mathcal{V} : \mathbb{F}_2^t \times \mathbb{F}_2^t \to \{\bot, \top\},\ (T, T') \mapsto \begin{cases} \top & \text{if } T = T' \\ \bot & \text{if } T \neq T' \end{cases}$$

and checks if the received tag T matches the computed tag T'. If they agree, then $\mathcal{V}$ returns the symbol $\top$ for success, and otherwise the symbol $\bot$ for failure.

There are many ways to construct MACs. A common approach is to take a cryptographic hash function and use it within the HMAC mode [30]. Another option are sponge functions [45] which are discussed further below.

1.1.5 Authenticated Encryption Schemes

Authenticated encryption (AE) schemes [32, 67] are an enhancement of common symmetric encryption algorithms and provide not only privacy of processed data but also ensure its integrity and authenticity. In other words, AE schemes try to achieve all of the three fundamental goals of symmetric cryptography introduced in the beginning of this section. *Authenticated encryption with associated data* (AEAD) [211] is an extension of AE that allows to process additionally so-called *associated data* (AD) which is not encrypted, i.e. it is transmitted in clear, but whose authenticity and integrity is ensured. Nowadays, AE(AD) schemes are the standard tool to protect in-transit data. AD can have many forms, like routing information in headers of datagram packets. Obviously, such a header (containing information like an IP address) has to remain unencrypted in order to be able to transmit the packet to the correct destination. Furthermore, the sender wants to ensure that the packet indeed reaches its destination and that a possible tampering with

the in-transit packet through Man-In-The-Middle attacks is detected. Finally, the receiver wants to be able to verify that the received data is from a valid source, namely the one with whom he exchanged secret keys. Below, we introduce the mathematical notation for AEAD and denote that AE is a special case of the prior where AD is left empty.

Let $k, n, t \geq 1$. An *authenticated encryption scheme with associated data* is a tuple $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$, where $\mathcal{K}$ is a key derivation function, $\mathcal{E}$ is an encryption and $\mathcal{D}$ a decryption function. The function $\mathcal{K}$ takes as input k and picks a secret key K uniformly at random from $\mathbb{F}_2^k$. We denote this operation by $K \xleftarrow{\$} \mathcal{K}(k)$ or just $K \xleftarrow{\$} \mathcal{K}$ if the context is clear. The encryption function is specified as

$$\mathcal{E} : \mathbb{F}_2^k \times \mathbb{F}_2^n \times \mathbb{F}_2^* \times \mathbb{F}_2^* \to \mathbb{F}_2^* \times \mathbb{F}_2^t,\ (K, N, A, M) \mapsto (C, T)$$

where K is a secret key, N a nonce, A associated data, M a plaintext message, C a ciphertext, and T an authentication tag. The decryption function, on the other hand, is defined by

$$\mathcal{D} : \mathbb{F}_2^k \times \mathbb{F}_2^n \times \mathbb{F}_2^* \times \mathbb{F}_2^* \times \mathbb{F}_2^t \to \mathbb{F}_2^* \cup \{\perp\},\ (K, N, A, C, T) \mapsto \begin{cases} M & \text{if } T = T' \\ \perp & \text{if } T \neq T' \end{cases}$$

where T' denotes the computed and T the received authentication tag.

Conceptually, a typical communication between two parties Alice and Bob is conducted as follows: assuming Alice and Bob have already exchanged a secret key K, Alice performs $\mathcal{E}_K(N, A, M) = (C, T)$ and sends the tuple (N, A, C, T) over the communication channel to Bob. Under the assumption that the AEAD scheme Π is secure, an adversary intercepting (N, A, C, T) can neither learn something about the message M from C or T nor can he modify any of N, A, C or T without being detected. In particular, he is not able to construct tuples (N', A', C', T') of his own that seem valid to Bob since he is not in possession of the shared secret key K. Bob, the valid communication partner, uses the decryption function $\mathcal{D}_K$ of Π on (N, A, C, T), which first verifies that the received authentication tag T is valid by comparing it with the computed tag T' and if so $\mathcal{D}_K$ returns the message M. If tag verification fails, $\mathcal{D}_K$ outputs nothing except for an error $\perp$ and securely erases all intermediate results. Now, we will give an overview on common AE(AD) constructions.

Generic Composition

There are several ways to construct AE(AD) schemes. A very common approach is *generic composition* [32], for which we give a brief overview in the following and discuss its pros and contras. Generic composition combines a symmetric encryption scheme, such as a block or stream cipher and a message authentication code (MAC) to form an AE(AD) scheme. Usually two different secret keys K_e and K_m are used for encryption $\mathcal{E}(K_e, \cdot)$ and authentication tag generation $\mathcal{T}(K_m, \cdot)$, respectively. To achieve AE using generic

composition there exist three well-known approaches which are discussed in more detail below.

Encrypt-and-MAC (EaM). The sender encrypts the message using the symmetric encryption algorithm, compresses the message using the MAC to obtain the tag and appends the tag to the ciphertext:

$$\mathcal{E}_{K_e}(M) \parallel \mathcal{T}_{K_m}(M) \ .$$

The receiver first decrypts the ciphertext to obtain the message and then uses the MAC on the message to verify the received authentication tag. Extending EaM to include associated data is straightforward:

$$A \parallel \mathcal{E}_{K_e}(M) \parallel \mathcal{T}_{K_m}(A \parallel M) \ .$$

The very well-known SSH protocol [248] is one representative that uses EaM-based schemes for authenticated encryption.

MAC-then-Encrypt (MtE). The sender compresses the message using the MAC, appends the generated authentication tag to the message and encrypts the result:

$$\mathcal{E}_{K_e}(M \parallel \mathcal{T}_{K_m}(M)) \ .$$

The receiver first decrypts the ciphertext, extracts message and tag, and then uses the MAC on the message to check if the received authentication tag is valid. The AEAD variant of MtE can be again constructed in the obvious way:

$$A \parallel \mathcal{E}_{K_e}(M \parallel \mathcal{T}_{K_m}(A \parallel M)) \ .$$

MtE-based authenticated encryption is used for example in (D)TLS [104], the protocol that enables secure communication on the Internet.

Encrypt-then-MAC (EtM). The sender encrypts the message to produce the ciphertext, uses the MAC on the ciphertext to produce the authentication tag and finally appends the tag to the ciphertext:

$$\mathcal{E}_{K_e}(M) \parallel \mathcal{T}_{K_m}(\mathcal{E}_{K_e}(M)) \ .$$

The receiver first checks if the received authentication tag is valid by using the MAC on the ciphertext and if so only then decrypts the ciphertext. The extension of EtM that includes associated data is specified as follows:

$$A \parallel \mathcal{E}_{K_e}(M) \parallel \mathcal{T}_{K_m}(A \parallel \mathcal{E}_{K_e}(M)) \ .$$

IPSec [150], the end-to-end security scheme operating on the IP layer of the *Internet Protocol Suite*, is using EtM to realise authenticated encryption.

Pros and Contras. Each of the above variants represents a valid approach to construct an AE(AD) scheme. However, one has to carefully consider which option to choose, because all of them have some disadvantages in one way or the other. We briefly discuss these issues below. From a security perspective only EtM satisfies all conditions for the construction of a secure AE scheme, see [32] for more details, and thus EtM is the only construction among the three that can be recommended without restrictions. All three variants can be found in real-world protocols and applications, though, as we have already seen above.

EaM obviously provides no integrity for the ciphertext since the authentication tag is computed from the plaintext. Moreover, an attacker could (theoretically) derive information on the plaintext from the MAC, for example, if the MAC only provides weak security. This issue is obviously avoided with EtM, where the tag is computed from the ciphertext. Another drawback of EaM-based schemes is the necessity to spend valuable resources on the decryption of the ciphertext before tag verification can be performed. If the latter fails then time spent on decrypting the ciphertext is wasted. In the worst case, this could lead to an increased vulnerability of applications to Denial-of-Service attacks, where an attacker floods a target with invalid ciphertexts trying to shut down the system through the overload. One well-known attack on the SSH protocol exploiting the above-mentioned EaM-weaknesses is described in [31].

MtE, the second variant, is from a theoretical perspective a better choice than EaM, see again [32]. From a practical point of view, though, it is unfortunately not ideal either and suffers from similar drawbacks as EaM: the ciphertext is not protected by the MAC and the authentication tag can only be verified after the ciphertext has been decrypted. Thus, resources spent on ciphertext decryption are wasted if tag verification fails. Additionally, the inclusion of the authentication tag into the ciphertext easily leads to security issues as exploited by so-called *padding oracle attacks* [237]. This attack basically allows an adversary to decrypt an entire message without the knowledge of the secret key if the *Cipher Block Chaining* (CBC) mode is used for data encryption. This is particularly problematic since CBC has been one of the standard block cipher modes used in a broad range of protocols. The problem can be traced back to flaws in the interaction between tag verification and plaintext expansion to the block length of the block cipher through the padding scheme.

EtM avoids the above problems since the authentication tag is computed from the ciphertext. The first step on the receiver's side is thus to verify the tag and only if it is valid, decrypt the ciphertext. On the one hand, this eliminates any potential danger to leak information on the plaintext through the MAC, and, on the other, allows to discard invalid ciphertexts much faster while no resources are wasted on the decryption of bogus messages as in the case of EaM and MtE. In summary, EtM not only has theoretical but also practical advantages over the other two variants and should be chosen if generic composition is considered for AE(AD). For example, the continued problems with MtE-based scheme in (D)TLS eventually led to discussions to replace the latter

with EtM-ciphers which promise much better security, performance, and robustness. The results of these discussions are summarised in [127].

All the generic composition-based schemes, however, share the drawback that two passes are required over the data, namely one for encryption and one for tag generation. While modern AE(AD) schemes derive very useful security features from the *two-pass* methodology, it is also often desirable to have *one-pass* AE(AD) solutions. This can be usually achieved through special AE(AD) block cipher modes or dedicated AE(AD) schemes. We discuss some of those options in the next sections below.

AE(AD) Block Cipher Modes

AE(AD) block cipher modes of operation allow to transform an arbitrary block cipher into an authenticated encryption scheme usually supporting associated data as well. In the following, we present a short overview on the most important ones.

Galois Counter Mode (GCM). GCM [196] is a one-pass nonce-based AE block cipher mode of operation supporting associated data. Its layout is suitable to achieve good speeds in soft- and hardware and can be parallelised for even higher performance. For example, GCM instantiated with AES, which is basically the default presently, achieves software speeds of 1.03 cycles per byte on the Intel Haswell micro-architecture [121], due to the availability of the special instructions AES-NI [122] and PCLMULQDQ [123]. In hardware, very high speeds (even beyond 100 Gbps) can be easily reached on FPGAs [190] or ASICs [189]. AES-GCM can be found, for instance, in TLS 1.2 [104] as an alternative to common MtE-based AEAD modes, in the IEEE 802.1ae media access control security (MACSec) standard [133], or in a number of industry cores [134, 179, 231].

Implementing AES-GCM is a rather complicated task, though, and constant-time implementations [149], necessary to thwart timing side-channel attacks [35], are even more challenging to realise without access to special CPU instructions, like AES-NI. Additionally, non-AES-NI constant-time implementations suffer from a noticeable performance-loss. Moreover, Joux presented an attack [136] showing that GCM is susceptible to forgery attacks if a nonce-key pair is repeated, essentially allowing an attacker to retrieve the secret key used for the computation of the authentication tag.

Offset Codebook Mode (OCB). Like GCM, OCB1-3 [171, 212, 214] is a one-pass nonce-based AE block cipher mode supporting a block size of 128 bit. It is usually instantiated with AES, where it achieves very good performance in soft- and hardware exceeding that of AES-GCM. For instance, AES-OCB runs at around 0.69 cycles per byte on the Haswell micro-architecture when AES-NI [122] instructions are used [121]. To achieve even greater speeds, OCB allows parallelization of data processing as well. Starting with version 2 [212], OCB also supports associated data making it effectively an AEAD scheme. OCB3 [171] introduced some minor changes regarding offset computation

and improved once more the performance of the scheme. A further advantage of OCB, when compared to GCM, is that it is much easier to implement, which also holds for constant-time implementations. Unfortunately, OCB never found wide-spread adoption due to patent restrictions. In 2013, Rogaway simplified licensing of OCB considerably, e.g. allowing free usage of the scheme in open-source software. Despite its many advantages it is also not completely without flaws. One minor problem pointed out in [111] is a collision attack that could be exploited if very large amounts of data are processed. In order to prevent this attack, the size of processed data per key has to be limited to about 64 GiB.

Counter with CBC-MAC (CCM). CCM [242] is an AE block cipher mode for block lengths of 128 bit. It is usually instantiated with AES and was meant to be an alternative for OCB avoiding the patenting issues of the latter. CCM combines CBC-MAC for authentication with CTR mode for encryption in a MAC-Then-Encrypt manner. CTR makes the scheme effectively a stream cipher that requires unique nonces for initialisation as long as the key is fixed. This is necessary, as confidentiality can not be guaranteed for CTR if nonces are repeated. A drawback of CCM is that it is not online, meaning, the length of the processed data has to be known in advance before one can proceed with encryption and thus processing of data streams is prevented. In [215] even more design-flaws are discussed, targeting different topics such as efficiency, parametrization, complexity, variable-tag-length subtleties, and wrong security claims. These all lead to the impression that CCM was not designed thoroughly. Despite these issues, CCM found its way into various protocols like IEEE 802.11i (WPA2), IPSec [150] and TLS 1.2 [104].

EAX Mode. EAX [33] is a nonce-based AEAD block cipher mode with no restrictions on the block length and supports authentication tag sizes up to the cipher's block size, which makes EAX very flexible. EAX was designed by the OCB team, aiming to address the many problems of CCM [215]. EAX has many desirable features: first of all it is accompanied with a proof of security showing that the security of the scheme can be reduced to the security of the underlying block cipher; ciphertext expansion is minimal, in the sense that the ciphertext has the same length as the plaintext plus the length of the authentication tag; CTR mode requires no decryption function per se, since encryption and decryption are done simply by XORing the plaintext and ciphertext with a stream of pseudo-randomly generated bits; it is an online algorithm capable of processing streams of data without the necessity to know the total length of data in advance; finally EAX can process static AD, which is for example useful when handling session data that changes only infrequently.

Sponge Functions

Many of the symmetric-key modes are based on block or stream ciphers, as we have already seen above, but there exist also modes that use a fixed-size permutation as the

underlying primitive. Designing such a permutation in a cryptographically strong way is, in some sense, equivalent to designing a block cipher without a key schedule. A very famous representative of these modes are the family of cryptographic sponge functions [45] which were introduced alongside of KECCAK [47] during the SHA-3 competition [222]. One of the remarkable features of sponge functions are their support for arbitrarily long input and output sizes which allows to build various kinds of primitives like hash functions, such as KECCAK, or stream ciphers.

Beyond that, sponge functions can also be used to construct authenticated encryption schemes supporting associated data. These variants are then better known as duplex constructions. We will focus in the following on this type, since NORX, the authenticated encryption scheme introduced in Chapter 4, is also based on a duplex construction. Regarding basic definitions and notation we let ourselves guide by the work of Bertoni et al. [48] which presents a comprehensive introduction to the topic. Besides the specification we give an overview on the most important properties of duplex constructions as well.

Duplex Constructions. Duplex constructions (and sponge functions) are defined over a fixed-length function f, a padding scheme pad, and a parameter r in bits. The function f is specified as

$$f : \mathbb{F}_2^b \to \mathbb{F}_2^b$$

with $b = r + c$ bits, where b, r, and c are called *width*, *rate* and *capacity*, respectively. The first r bits of the state are used for data processing while the last c bits ensure the security of the primitive and are never affected directly by the input blocks or returned as output. Although not essential, the function f is usually chosen to be a permutation on b bits, which gives better security properties in general. The second component of a duplex construction, the padding rule

$$\mathsf{pad}_r : \mathbb{F}_2^n \to \mathbb{F}_2^{rm}$$

extends an n-bit string X to a multiple of the rate r, which is necessary for processing data of arbitrary sizes. In order to guarantee security, such a padding scheme has to be *sponge compliant* [45], which means that it must be injective, non-empty, and has to ensure that the last block is non-zero. We assume in the following that all input data has been padded accordingly and write $X = X_0 \parallel \cdots \parallel X_{m-1}$ with $|X_i| = r$ for $0 \leq i \leq m-1$.

While sponge functions are stateless in between calls, a duplex construction accepts, after initialisation, calls that take as input a bit string X_i and a requested number of output bits l_i, with $0 \leq l_i \leq r$, and returns an l_i-bit sized output string Y_i such that the latter depends on all X_j for $0 \leq j \leq i$. In other words, an output of a duplex construction depends on all the inputs received so far. The process detailed above is also called *duplexing* and is denoted by

$$Y_i = D.\mathsf{duplexing}(X_i, l_i)$$

where D denotes a *duplex object*, which is a concrete instance of a duplex construction. Internally, first the input block X_i is XORed into the first r bits of the state, then the function f is applied to the latter, and finally the first l_i bits of the state are extracted and returned as output. Figure 4 shows the layout of a generic duplex construction.

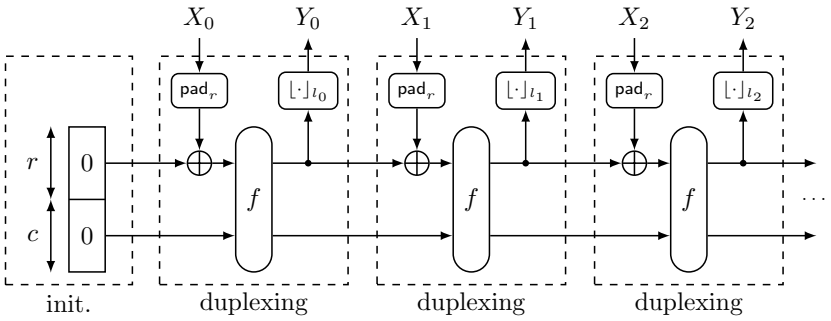


Figure 4: The duplex construction.

Designing an authenticated encryption scheme with support for associated data from a duplex construction can be achieved as follows. First, the state is initialised with 0^b, followed by absorption of a secret key K and a nonce N in the first duplexing call. Usually, no output is produced in this phase. Depending on the concrete sizes of r, N, and K, multiple duplexing calls might be necessary until all of the data has been absorbed. After this initial setup-phase, one can start processing actual data.

Now, let $A = A_0 \parallel \cdots \parallel A_{a-1}$ denote the (already padded) associated data, i.e. $|A_i| = r$ for $0 \leq i \leq a-1$, and let $M = M_0 \parallel \cdots \parallel M_{m-1}$ denote the (already padded) message, i.e. $|M_j| = r$ for $0 \leq j \leq m-1$. Without loss of generality, we assume that A is processed before M, but it is also allowed to be the other way round. In fact, duplex constructions enable to process arbitrarily interleaved data of different types, but we omit this case here for reasons of simplicity.

Authentication of associated data is done by calling D to absorb block A_i without requesting any output bits. Thus, the call is of the form $D.\mathsf{duplexing}(A_i, 0)$ for $0 \leq i \leq a-2$. Finally, during the duplexing of the last block A_{a-1} of associated data, r output bits are requested, i.e. $Y_{-1} = D.\mathsf{duplexing}(A_{a-1}, r)$, which are then used to encrypt the first plaintext block M_0 and obtain the corresponding ciphertext block via $C_0 = Y_{-1} \oplus M_0$. During plaintext processing r bits of output are requested in each call $Y_j = D.\mathsf{duplexing}(M_j, r)$ and the ciphertext blocks are obtained by $C_j = Y_{j-1} \oplus M_j$ for $0 \leq j \leq m-2$. Then again, the last block of plaintext processing is handled differently, by requesting t instead of r output bits which are used as the authentication tag T, or, in other words, the last call is equivalent to $T = D.\mathsf{duplexing}(M_{m-1}, t)$. This finishes authentication of A and authenticated encryption of M and the tuple (N, A, C, T) can be transmitted.

Properties of Duplex Constructions. Authenticated encryption modes based on duplex constructions have many desirable properties:

- Duplex constructions inherit all the strong security bounds of the sponge function family and benefit from the extensive analysis conducted on sponge functions [7, 43, 45, 46, 48, 51, 141].
- Encryption is performed like in a stream cipher namely by XORing the plaintext with a pseudo-randomly generated key stream, which allows to perform decryption analogously. Thus, the function f is sufficient for both encryption and decryption and no inverse function f^{-1} is necessary.
- Data that requires authentication and data that requires authenticated encryption can be interleaved arbitrarily.
- Duplex constructions can issue intermediate tags due to their flexible data processing capabilities.
- Encryption is not expanding, i.e. plaintext and ciphertext have the same length.
- Duplex constructions are single-pass and require only one call to the function f for every processed data block.

There are also some limitations, though. Firstly, the basic variant of the mode is serial and cannot be parallelized on an algorithmic level. Nevertheless, in Chapter 4 we will introduce a modified version of the duplex construction for NORX which is capable of processing data in parallel. Secondly, since encryption works like in a stream cipher, it is essential for the security of the scheme that the nonce freshness is guaranteed. Otherwise, the first differing plaintext blocks $M \neq M'$ that are encrypted with the same key stream block Y leak their respective XOR through the XOR of the corresponding ciphertexts C and C', namely $C \oplus C' = (M \oplus Y) \oplus (M' \oplus Y) = M \oplus M'$.

Other AE Constructions

There are also AE(AD) schemes following other design approaches that do not fall into one of the aforementioned categories. For example, Helix [113], Phelix [243], and Hummingbird-2 [106] are dedicated hybrid AE primitives offering efficient stream encryption and MAC computation at the same time, similar to the duplex construction described above. However, all three of these primitives were shown to be weak [192, 201, 203, 247]. Another example is the stream cipher Grain-128a [2] which offers optionally an extension for authenticated encryption. At this point, we do not go into further details but refer the interested reader instead to the referenced literature.

1.2 Cryptanalysis

How secure is a given cryptographic construction? The main goal of cryptanalysis is to find the answer to this questions. There are countless ways how a given cryptographic primitive can be analysed. In the following, we introduce the general categories of cryptanalytic attacks. An overview is given in Figure 5 which is of course neither exhausting nor exact in every detail.

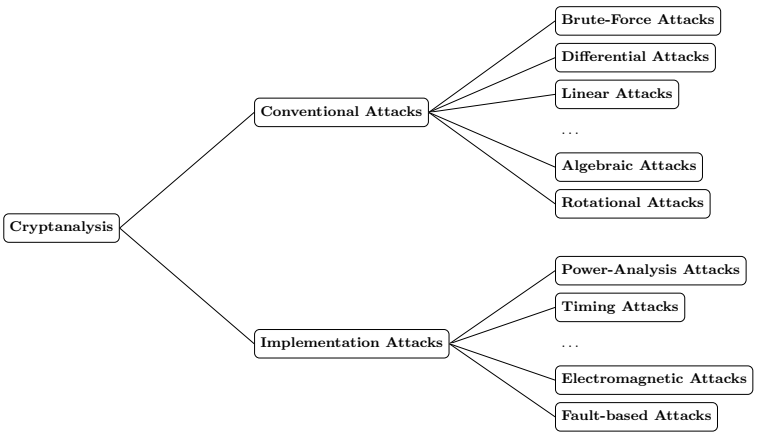


Figure 5: Categories of common cryptanalytic attacks.

The success of an attack is usually measured in terms of required *time*, *memory*, and *data*. It usually depends on two factors, namely on the *attack outcomes*, which categorises the goals an adversary tries to achieve with his attack, and the *adversarial model*, which specifies what an adversary is allowed or capable of doing during an attack. Further, an attack against a class of cryptographic constructions is called *generic*, if it works without exploiting any concrete details of the members of that class. For example, exhaustively searching through all candidates of the key space of a symmetric-key primitive is a generic attack. Otherwise, if an attack requires certain features of a concrete cryptographic construction, it is called *non-generic*.

In the field of cryptanalysis, one usually assumes that the adversary knows all the details of the attacked cryptographic primitive except for the secret key that was supplied by the user. This assumption is also known as *Kerckhoffs' Principle*, which dates back to Auguste Kerckhoffs who laid out requirements for a usable field cipher in 1883 [151].

Time, Memory and Data

In cryptanalysis, the success of an attack is measured according to the amount of resources it consumes. As already noted above, there are usually three types of resources that are

interesting for an attack:

- **Time.** The time, or work effort, required to mount an attack. How time is measured concretely, often depends on the given attack. One example is the amount of necessary encryption operations. Time is usually the basic resource by which the effectiveness of an attack is categorised but it is not the only one. Time is also often referred to as the *offline complexity* of an attack.
- **Memory.** The required amount of memory to execute an attack is in many cases another important factor. Generally, if an attack has a high memory consumption it is far more worse than having the same amount of time consumption. There exists a rule of thumb saying that "time is cheaper than memory" [128], which nicely captures the intuition that, for example, when given a 128-bit block cipher, it is easier to perform 2^{40} encryption runs than storing 2^{40} encryption results ($= 16\,\text{TiB}$) in memory.
- **Data.** The required amount of data is the third important resource of an attack. If the time required to retrieve the data for an attack far exceeds normal usage patterns, then the practical impact of the attack is limited. Data is also often referred to as the *online complexity* of an attack.

Note that attacks usually do not require just a single one of the above resources but rather a combination of all three. Resources can be also traded against each other, which leads to so-called *trade-off attacks*, a concept which is briefly discussed later.

The amount of consumed resources adds another dimension to the categorisation of an attack: if it is not feasible to raise the required resources in a practical context (with current technology), then the attack is called *theoretical*. Otherwise, it is denoted *practical*. For example, an attack that targets a 128-bit block cipher and that requires 2^{120} encryption runs, can obviously be categorised as theoretical. In contrast, an attack on the above cipher that requires 2^{40} encryption runs (and negligible other resources), can certainly be considered practical.

Attack Objectives

The possible results of an attack can differ greatly and depend on various factors. A somewhat simplified categorisation of attack objectives for block ciphers, as introduced in [162], can be given as follows:

1. **Key Recovery.** The attacker is able to recover the secret key K. This is the most powerful result of an attack.
2. **Global Deduction.** The attacker is able to compute encryption $\mathcal{E}_K(\cdot)$ or decryption $\mathcal{D}_K(\cdot)$ without knowing the secret key K.

3. **Local Deduction.** The attacker can compute encryption $\mathcal{E}_K(M)$ or decryption $\mathcal{D}_K(C)$ without knowing K for some messages M or ciphertexts C.

4. **Distinguishing.** The attacker can effectively distinguish $\mathcal{E}_K(\cdot)$ from a permutation chosen uniformly at random. Trying to distinguish encrypted from random data is the most basic attack an adversary can mount on a cryptographic primitive.

These attack outcomes are ordered such that an adversary achieving one of them automatically achieves all that follow. This means in particular if an attacker is not capable of distinguishing a given block cipher from a permutation chosen uniformly at random, then the block cipher is, in some sense, ideal.

Note that the above hierarchy of attack outcomes might differ for other cryptographic primitives. For example, an attack objective for stream ciphers might be the reconstruction of the internal state, which is obviously a very powerful result, but does not necessarily lead directly to recovery of the secret key. Adversaries targeting keyless hash functions have yet again differing attack objectives. While distinguishing a hash function from a pseudorandom function still forms the basis in this context, the objective of key recovery is obviously meaningless. Instead, attackers are usually interested in constructing collisions, pre-images, or second pre-images, see Section 1.1.3.

Adversarial Models

The capabilities of an adversary in terms of operations he is able or allowed to execute, is another important factor during a cryptanalytic attack. These conditions are commonly summarised in *adversarial models* and are categorised by the type of data and by the type of access an adversary requires to successfully mount a given attack. The type of data differentiates between inputs and outputs of a cryptosystem such as secret keys, plaintexts, and ciphertexts, and the type of access differentiates between reading, writing and adaptive writing access, which are denoted as known values, chosen values and adaptively chosen values, respectively. An overview on the main adversarial models in conventional cryptanalysis, as presented in [162], is given below:

1. **Ciphertext-only Attacks.** The adversary knows only the ciphertext and has no access to the plaintext. A cryptographic primitive vulnerable to such kind of attacks is considered exceptionally weak, since it is possible to distinguish it from a random permutation by analysing only ciphertexts.

2. **Known-plaintext Attacks.** The adversary has reading access to plain- and corresponding ciphertexts processed by the cipher. A representative of this category is, for example, linear cryptanalysis [182].

3. **Chosen-plaintext Attacks.** These are similar to known-plaintext attacks, with the difference that an adversary is allowed to choose the concrete plaintexts to

be encrypted prior to the attack. A well-known attack type of this category is differential cryptanalysis [58].

4. **Chosen-ciphertext Attacks.** The adversary can choose ciphertexts to be decrypted by the cipher before the attack starts and has reading access to the resulting plaintexts.

5. **Adaptively Chosen-plaintext Attacks.** The adversary can select plaintexts to be encrypted during the attack and is not forced to choose them before the attack starts as in the case of the chosen plaintext scenario. The attacker also has access to the resulting ciphertexts.

6. **Adaptively Chosen-ciphertext Attacks.** The adversary can select ciphertexts to be decrypted during the attack and is not forced to choose them before the attack starts as in the case of the chosen ciphertext scenario. The attacker also has access to the resulting plaintexts.

7. **Related-key Attacks.** The adversary can encrypt plaintexts and decrypt ciphertexts with the attacked key and with keys related to the latter [53], which, for example, differ only at certain bit positions.

The attacker has more control over the analysis of the block cipher with each of the above steps and allows him to create increasingly powerful attacks. However, at the same time collecting data of a given type becomes more and more demanding the further we go down that list. The above categorisation also presents an indication on the (im-)practicability of the attacks.

The above models also form the basis for implementation attacks, however, an attacker is assumed to have additional capabilities. Concrete details are discussed later in this chapter.

1.2.1 Brute-Force Attacks

A conceptually very simple attack, operable against any symmetric cryptographic primitive, is an exhaustive search for the shared secret key K. Obviously, this approach is independent of the design of the cipher. For example, in the case of block ciphers, the adversary simply enumerates all key candidates of the search space and tests every single one of them against a known message-ciphertext pair until the correct key is found. This particular category of cryptanalytic techniques is also known as *brute-force attacks*. Since there is no way of preventing an adversary from mounting such an *exhaustive search*, designers of cryptographic primitives try to ensure that brute-force is the best attack available to an adversary. Exhaustive search techniques are also often part of more advanced cryptanalytic attacks.

More formally, an attacker who knows a message-ciphertext pair $(M, \mathcal{E}_K(M))$ and the corresponding encryption algorithm $\mathcal{E}$, "just" needs to try 2^k keys to find the secret key K with probability one, where $k = |K|$. In general, if he checks $n \leq 2^k$ keys, he succeeds with a probability of $n/2^k$ and if he targets $m < n/2^k$ keys at once, he succeeds with a probability of $mn/2^k$.

Time-Memory Trade-Offs

Exhaustive search techniques test one key after another but take no role for memory into account. However, in many cases it is possible to improve certain attacks if some form of memory is available which also holds for brute-force. An obvious application of memory in cryptanalysis of block ciphers are so-called *dictionary attacks* which are, in some way, the counterpart to exhaustive search. In the *offline phase*, i.e. before the actual attack starts, an attacker pre-computes all 2^k possible ciphertexts for a single known plaintext and stores all of the key-ciphertext tuples in a table sorted by the value of the ciphertext. If an adversary then intercepts a ciphertext in the *online phase* of the attack, he just needs to look up the ciphertext thereby retrieving the corresponding key which represents a candidate for the secret key. The requirements for such dictionary attacks are 2^k *words* of storage where the size of a word depends on the attacked cipher. These two extreme situations, i.e. exhaustive search versus dictionary attacks, call out for a trade-off.

Time-memory trade-offs (TMTO) were introduced in the context of cryptanalysis by Hellman [128] in 1980. The idea here is simple: if a certain attack has to be carried out multiple times, it may be possible to execute the exhaustive search in advance and store all results in memory. In other words, the values pre-computed in the offline phase are used to improve the running time of the attack in the online phase. However, the storage requirements compared to a dictionary attack are greatly reduced. The typical application of this method is the recovery of a key K when a plaintext M and its corresponding ciphertext $C = \mathcal{E}_K(M)$ are known. The basic idea of Hellman's TMTO attack is to compute from a chosen plaintext M and a sequence of key candidates $K_{0,0}, \ldots, K_{s-1,0}$, the starting points, key sequences $K_{i,j+1} = R(\mathcal{E}_{K_{i,j}}(M))$ of length t, with $i \in \{0, \ldots, s-1\}$ and $j \in \{0, \ldots, t-2\}$, where R is a reduction function that maps a ciphertext to a key candidate. From those sequences only the starting and end points are saved in a table as pairs $(K_{0,0}, K_{0,t-1}), \ldots, (K_{s-1,0}, K_{s-1,t-1})$. Once an attacker intercepts a ciphertext C he can use the precomputed tables to check for potential key candidates by going step-wise through the table partially reconstructing intermediate results of the t key sequences if no match is found. Without going into the exact details at this point, the work in [128] shows that, for a cryptosystem with 2^n keys, the secret key can be recovered in $2^{2n/3}$ operations and $2^{2n/3}$ words of memory. To put Hellman's TMTO attack into perspective, it is estimated that the above attack can be used against DES requiring approximately 64 GiB of memory and 2^{48} DES operations instead of 2^{56} DES operations for exhaustive search (in the worst case scenario).

Over the years many enhancements were published for Hellman's TMTO attack. In 1992, Rivest introduced *distinguished points* [102] where only key candidates of a particular shape are saved as end points in the table, like keys that only have zeroes in the ten leftmost bits. This approach offers a couple of advantages over normal Hellman tables. The structural knowledge can be used to reduce the number of memory accesses in the online phase of the attack. However, distinguished points also have some disadvantages. For example, it is harder to estimate the actual key coverage since the computed sequences are very likely to have different lengths. For a more detailed discussion we refer the interested reader to the literature [102, 162].

In 2003, Oechslin [200] introduced another improvement, the so-called *rainbow tables* which resolve some issues of Hellman's work and additionally offer computational benefits in the online phase. The most significant change is that rainbow tables use a sequence of reduction functions $R_1, \ldots, R_l$ instead of just a single one, which gives them advantages similar to distinguished points while avoiding their weaknesses. An attack based on rainbow tables requires about half the online work effort compared to an attack based on Hellman's tables while both attacks have the same key coverage, and requirements with respect to precomputation and memory. Rainbow tables became widely known through their application in password-cracking. A comprehensive coverage of the topic is beyond the scope of this doctoral thesis. Hence, we refer the interested reader again to the literature [162, 200].

Time-Processor Trade-Offs

One great advantage of brute force cryptanalysis is that it is trivial to parallelise. An attack that checks n keys can be simply distributed to c nodes, lowering the workload on each to n/c guesses. Obviously, parallelisation offers a linear speedup to brute-force attacks.

The design of dedicated hardware for parallel cryptanalytic attacks, so-called *time-processor trade-offs*, was discussed by Bernstein [38] in 2005. The work shows by estimation that a decently designed parallel machine can be much more efficient than a serial counterpart and the parallel machine being only about twice as expensive. Using AES as an example, the work illustrates that a parallel machine consisting of 2^{32} AES circuits and a comparable amount of memory has a probability of success of about 2^{-69} to find an AES key in only 2^{27} AES computations. If more than one key is targeted, say 2^{10} keys, then the probability increases to 2^{-59} but the number of required AES computations remains fixed at 2^{27}. For comparison, the serial machine is expected to find a single secret key respectively one out of 2^{10} keys with probabilities of 2^{-69} and 2^{-59} where both scenarios require 2^{59} AES computations. In other words, the parallel machine is by a factor of about $2^{59}/2^{27} = 2^{32}$ more efficient than its serial counterpart. In the context of brute-force attacks, time-processor trade-offs are (conjecturally) much more efficient than a time-memory trade-off, since the latter often neglects the communication cost between a

processor and a large memory. In comparison, the time-processor trade-offs above assume that each circuit has its own small memory where only a couple of intermediate results (about 2^4) are buffered, i.e. memory accesses are kept at the bare minimum. For further reading we also refer to the work of Wiener from 2004 [244] where he presented a survey on the true costs of cryptanalytic attacks.

1.2.2 Differential Attacks

Differential cryptanalysis was discovered by Biham and Shamir in the early 1990s [58, 59] where they investigated differential attacks on various block ciphers and hash functions. They noted, in particular, that DES seems to be remarkably resistant against differential attacks and would be much more vulnerable with only a few minor modifications. Coppersmith, who is one of the original designers of DES, published a paper in 1994 revealing that the IBM design team of DES had been aware of differential cryptanalysis as early as 1974 [88]. Differential cryptanalysis belongs to the most powerful tools in the repertoire of every cryptanalyst and, despite being invented for the cryptanalysis of block ciphers [162], was extended to other symmetric primitives as well [124, 187, 233, 247].

The basic idea of differential attacks is the exploitation of correlations between input and output differences of a cryptographic primitive, i.e. differential attacks utilise non-ideal propagation of differences in a primitive when considering plaintext-ciphertext pairs. Differences are usually computed with respect to bitwise `XOR`, but there are also other use cases where differences are considered, for example, with respect to modular integer addition. Differential cryptanalysis belongs to the category of chosen-plaintext attacks as introduced above.

In the simplest case, consider a cryptosystem very similar to a one-time pad which encrypts a plaintext M with a key K to a ciphertext C by computing $C = M \oplus K$. If K is used a second time to encrypt another message M', i.e. $C' = M' \oplus K$, then an attacker who intercepts both C and C' is able to trivially derive information on the plaintexts by computing the `XOR`-difference of the ciphertexts

$$C \oplus C' = (M \oplus K) \oplus (M' \oplus K) = M \oplus M' \ .$$

Although this is a very simple example, it nevertheless illustrates the basic idea of differential cryptanalysis very well. Since real-world ciphers are much more complex than the above example, a more general approach to differential cryptanalysis is required.

Differences and Differentials

In this part, we introduce the basic notions and concepts that are used in differential cryptanalysis.

Let $x, x' \in \mathbb{F}_2^n$ be n-bit strings. We call $\alpha = x \oplus x'$ the *n-bit difference of x and x' with respect to bitwise `XOR`* or just *`XOR`-difference* in short. For an n-bit difference α with

Hamming weight $\mathsf{hw}(\alpha) = m$, we call the m 1-entries of α also the *active bits* of α. Let f be a vector Boolean function of the form

$$f : \mathbb{F}_2^n \longrightarrow \mathbb{F}_2^m,\ x \mapsto y$$

with $n, m \in \mathbb{N}$ and let $\alpha \in \mathbb{F}_2^n$ and $\beta \in \mathbb{F}_2^m$ be XOR-differences. We call (α, β) an ***XOR-differential*** with respect to f, if there exists a bit string $x \in \mathbb{F}_2^n$ such that the following equation holds:

$$f(x \oplus \alpha) \oplus f(x) = \beta\ .$$

If no such bit string x exists, then (α, β) is called an *impossible* XOR-differential with respect to f. We denote a differential by

$$\alpha \xrightarrow{f} \beta\ .$$

If the context is clear we skip the f above the arrow and just write $\alpha \longrightarrow \beta$. Furthermore, we call α the *input difference* and β the *output difference* of the differential with respect to the function f.

Each differential has an associated probability, which describes the likelihood that, for input pairs x and $x \oplus \alpha$ where x was chosen uniformly at random, the output difference β indeed appears after the application of f. Let f be a vector Boolean function as specified above and let $\delta = (\alpha, \beta)$ be an XOR-differential with respect to f. The probability xdp^f that δ holds is defined as

$$\mathsf{xdp}^f(\delta) = |\{x \in \mathbb{F}_2^n : f(x \oplus \alpha) \oplus f(x) = \beta\}| \cdot 2^{-n}\ .$$

The value $\mathsf{xdp}^f(\delta)$ is also called the ***XOR-differential probability*** of δ. Moreover, for $\mathsf{xdp}^f(\delta) = 2^{-w}$ we call w the ***XOR-(differential) weight*** of δ.

Note that the differential probability of an impossible differential is always 0 by definition, since $\{x \in \mathbb{F}_2^n : f(x \oplus \alpha) \oplus f(x) = \beta\} = \emptyset$. To capture all information on a differential (α, β) of f having probability p in a compact form, we write

$$\alpha \xrightarrow[p]{f} \beta\ .$$

Differential cryptanalysis was originally developed for the security analysis of block ciphers as already mentioned in the introduction. These cryptographic primitives are usually built from a (cryptographically weak) round function f which is then iterated r times. However, for decently designed block ciphers, it is usually infeasible to directly find differentials of high probability for all r rounds. Therefore, it is reasonable to not only consider input and output differences of the cryptographic primitive but to analyse intermediate values after each of the r rounds as well. This leads to the concept of

differential characteristics (or *paths*, or *trails*). Let $f_0, \ldots, f_{r-1}$ be a sequence of vector Boolean functions defined by

$$f_i : \mathbb{F}_2^n \longrightarrow \mathbb{F}_2^n,\ x \mapsto y$$

for $i \in \{0, \ldots, r-1\}$ and let $\alpha_0, \ldots, \alpha_r \in \mathbb{F}_2^n$ denote differences such that

$$\alpha_i \xrightarrow{f_i} \alpha_{i+1} \ .$$

We call $(\alpha_0, \ldots, \alpha_r)$ a *(XOR-differential) characteristic*, or *path*, or *trail* with respect to the functions $f_0, \ldots, f_{r-1}$ and denote it by

$$\alpha_0 \xrightarrow{f_0} \ldots \xrightarrow{f_{i-1}} \alpha_i \xrightarrow{f_i} \ldots \xrightarrow{f_{r-1}} \alpha_r \ .$$

The values α_0 and α_r are called the *input-* and *output difference* and α_j with $j \in \{1, \ldots, r-1\}$ are called the *internal differences* of the characteristic.

A visualisation of such a differential characteristic in an iterated block cipher with rounds f_i and round keys K_i is given in Figure 6 for $i \in \{0, \ldots, r-1\}$.

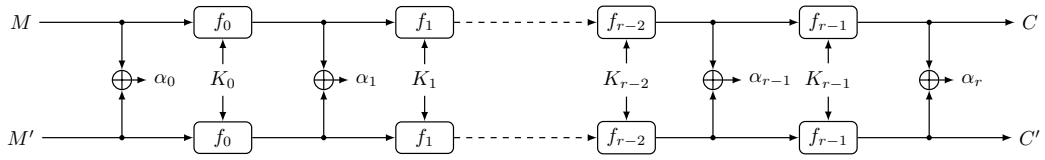


Figure 6: XOR-differential characteristic in an iterated r-round block cipher.

To compute the differential probability p of the entire characteristics, one generally assumes that the sequence of differences forms a Markov chain and that the plaintexts and round subkeys are independent and uniformly random [177]. Thus, p is simply the product of the probabilities of each single step. More formally, let $(\alpha_0, \ldots, \alpha_r)$ be a differential characteristic with

$$\alpha_i \xrightarrow[p_i]{f_i} \alpha_{i+1}$$

where $p_i = \mathsf{xdp}^{f_i}(\alpha_i, \alpha_{i+1})$ for $i \in \{0, \ldots, r-1\}$. The overall probability p of the characteristic $(\alpha_0, \ldots, \alpha_r)$ is then approximated by

$$p \approx \prod_{i=0}^{r-1} p_i \ .$$

An obvious question that comes to mind at this point is how differentials and characteristics relate to each other. Differentials can be composed of multiple differential

characteristics which share the same input and output differences α_0 and α_r, respectively, but have distinct internal differences α_i for $i \in \{1, \ldots, r-1\}$. In a first step, it is often assumed that the probability of a differential can be approximated by the highest probability of one of its differential characteristics. While it works in most cases as an initial approximation, the latter usually turns out to be too rough due to differential effects such as *trail clustering* [49, 64] where many characteristics with a similar probability and the same input and output differences form a differential and equally contribute to its probability. As a consequence, the probability of the differential is much higher than that of the single characteristics.

XOR-differentials are the most common type used in differential cryptanalysis. However, one could transfer the above concepts to other group operations and their inverses, too. One such class are, for example, f-differentials with respect to XOR where f is a vector Boolean function. We briefly motivate this approach below. Assume that differences are expressed through a vector Boolean function

$$f : \mathbb{F}_2^{2n} \longrightarrow \mathbb{F}_2^n$$

instead of XOR. A tuple (α, β, γ) of differences is called an *f-differential* with respect to XOR, if there exist n-bit strings x and y such that the following equation holds:

$$f(x, \alpha) \oplus f(y, \beta) = f(x \oplus y, \gamma) \ .$$

If no such n-bit strings x and y exist, the f-differential is called *impossible* with respect to XOR. We denote such an f-differential by $(\alpha, \beta) \longrightarrow \gamma$, where α and β are the *input differences* and γ is the *output difference*.

Let f be a vector Boolean function and δ be a f-differential. The probability $\mathsf{fdp}^{\oplus}$ that δ holds is defined as

$$\mathsf{fdp}^{\oplus}(\delta) = |\{x, y \in \mathbb{F}_2^n : f(x, \alpha) \oplus f(y, \beta) \oplus f(x \oplus y, \gamma) = 0\}| \cdot 2^{-2n} \ .$$

We call $\mathsf{fdp}^{\oplus}(\delta)$ the *f-differential probability* of δ. Moreover, for $\mathsf{fdp}^{\oplus}(\delta) = 2^{-w}$ we call w the *f-(differential) weight* of δ.

The notions of an f-differential characteristic and its associated probability can be defined analogously to those of XOR-differential characteristic above.

Using Differentials

Differentials can be used for cryptanalysis in various ways. Below we review the most common applications, which are *distinguishers*, *key recovery*, and construction of *collisions* in hash functions.

Distinguishers. Let $\mathcal{E}$ be a block cipher with a k-bit key and an n-bit block size and let (α, β) be a differential for $\mathcal{E}$ having probability $p \gg 2^{-n}$, where $\gg$ means "significantly larger". A simple application of (α, β) is to mount a *distinguishing attack* on $\mathcal{E}$, i.e. an attack that tries to distinguish $\mathcal{E}$ from an ideal cipher. For more information, refer to the attack objectives as introduced at the beginning of the current section. The sketch of such an attack is given in Algorithm 1. It takes as input the above differential and an encryption oracle $\mathcal{O}_{\mathcal{E}_K}$, which, when queried with a plaintext M, returns the corresponding ciphertext C. Note that the secret K is unknown to the attacker and is assumed to remain fixed for the duration of the experiment. Then two oracles are queried $1/p$ times with randomly chosen messages M and $M \oplus \alpha$ and it is checked if the outputs $C = \mathcal{O}_{\mathcal{E}_K}(M)$ and $C' = \mathcal{O}_{\mathcal{E}_K}(M \oplus \alpha)$ exhibit the required output difference, i.e. if $C \oplus C' = \beta$. If, at some point, such a match is found the distinguisher returns `false`, meaning that $\mathcal{E}$ is not ideal. If no match is found it returns `true`. The attack is expected to succeed with probability close to 1, since $1/p$ checks are executed before two plaintext-ciphertext pairs that match the differential are found. In contrast, for an ideal cipher it is expected that about 2^{n-1} trials are necessary.

Algorithm 1: distinguish$((\alpha, \beta), \mathcal{O}_{\mathcal{E}_K})$

Inputs:
differential (α, β) for $\mathcal{E}$ of probability p, encryption oracle $\mathcal{O}_{\mathcal{E}_K}$
Outputs:
{`true`, `false`}
Algorithm:
1. **for** $i \in \{0, \ldots, 1/p - 1\}$ **do**
2. $\quad M \xleftarrow{\$} \mathbb{F}_2^n$
3. $\quad$ **if** $\mathcal{O}_{\mathcal{E}_K}(M) \oplus \mathcal{O}_{\mathcal{E}_K}(M \oplus \alpha) = \beta$ **then**
4. $\quad\quad$ **return** `false`
5. $\quad$ **end**
6. **end**
7. **return** `true`

Key Recovery. The ultimate aim of an attacker is not only to distinguish a cipher from a random permutation but to recover the secret key. A distinguishing attack can be converted into a key recovery attack as follows: Suppose the encryption $\mathcal{E}$ of an n-bit block cipher is composed of round functions f_i for $i \in \{0, \ldots, r-1\}$, uses n-bit sub keys K_j for $j \in \{0, \ldots, r\}$, with $K = K_0 \parallel \cdots \parallel K_r$, and can be written as

$$C = \mathcal{E}(K, M) = f_{r-1}(f_{r-2}(\ \ldots \ f_1(f_0(M \oplus K_0) \oplus K_1) \ \ldots \ \oplus K_{r-2}) \oplus K_{r-1}) \oplus K_r \ .$$

In other words, $\mathcal{E}$ can be modelled as shown in Figure 7.

Further assume that an attacker found a differential (α, β) of probability p stretching over the first $r-1$ rounds $f_{r-2} \circ \cdots \circ f_0$. The attacker initialises counters $T_0, \ldots, T_{n-1}$

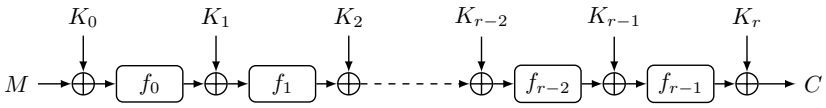


Figure 7: The encryption function $\mathcal{E}_K$ of a block cipher.

with the value 0 each. For randomly chosen plaintexts M and $M' = M \oplus \alpha$ he then queries the encryption oracles to obtain $C = \mathcal{O}_{\mathcal{E}_K}(M)$ and $C' = \mathcal{O}_{\mathcal{E}_K}(M')$. Afterwards, he iterates over all possible values $k \in \{0, \ldots, 2^{n-1}\}$ of K_r and checks if

$$\beta = f_{r-1}^{-1}(C \oplus k) \oplus f_{r-1}^{-1}(C' \oplus k) \ .$$

If the equation holds then the counter T_k is increased by 1. This event occurs with probability p if the kth guess for K_r is correct and with another probability p' if it is incorrect. It is expected that $p \gg p'$, i.e. that p is much larger than p'. If the above experiment is repeated for $l \gg 1/p$ randomly chosen message pairs M and $M \oplus \alpha$ then the correct counter is expected to have a value of approximately $l \cdot p$ whereas the counters for the incorrect key hypotheses have values of approximately $l \cdot p'$. Since $p \gg p'$, it follows that $l \cdot p \gg l \cdot p'$, i.e. the counter for the correct key should be clearly distinguishable from the counters of the other key hypotheses and the attacker can thus reconstruct K_r. Afterwards, the last round can be stripped off and K_{r-1} can be attacked by a similar technique using a differential over $r-2$ rounds. Obviously, the attacker can repeat this approach until he has retrieved all subkeys. A sketch of the above key recovery attack is shown in Algorithm 2. Obviously, the described attack can be transferred to other cipher constructions as well.

The so-called *signal-to-noise ratio* $S_N = p/p'$ measures the quality of a differential attack and is used to describe the advantage of the differential attack over exhaustive search [220]. Instead of trying to rank the correct n-bit key as the most significant one, as shown in Algorithm 2, one instead tries to rank it within the top m out of 2^n key candidates. We then say that the attack yields a $\log_2 m$-bit advantage over exhaustive search, i.e. the complexity is reduced by a factor of $2^{n-\log_2 m}$. Assuming that the key counters $T_0, \ldots, T_{n-1}$ are independent and that they are identically distributed for all wrong key candidates, one can compute the probability of success p_s for a differential attack using N plaintext-ciphertext pairs as follows

$$p_s = \Phi\left(\frac{\sqrt{pNS_N} - \Phi^{-1}(1 - 2^{-\log_2 m})}{\sqrt{S_N + 1}}\right)$$

where Φ is the cumulative distribution function of the standard normal distribution [220]. The work [220] also shows, how to reformulate this result such that, given the targeted probability of success p_s, one can compute the corresponding data complexity of the

Algorithm 2: recover_key$((\alpha, \beta), \mathcal{O}_{\mathcal{E}_K})$

Inputs:
$(r-1)$-round differential (α, β) of probability p, encryption oracle $\mathcal{O}_{\mathcal{E}_K}$
Outputs:
round key K_r
Algorithm:

```
1.  (T_0, ..., T_{n-1}) ← (0, ..., 0)
2.  for i ∈ {0, ..., l-1} do
3.      M ←$ F_2^n
4.      C ← O_{E_K}(M)
5.      C' ← O_{E_K}(M ⊕ α)
6.      for k ∈ {0, ..., n-1} do
7.          if f_{r-1}^{-1}(C ⊕ k) ⊕ f_{r-1}^{-1}(C' ⊕ k) = β then
8.              T_k ← T_k + 1
9.          end
10.     end
11. end
12. T_j ← max(T_0, ..., T_{n-1})
13. return j
```

differential attack:

$$N = \frac{(\sqrt{S_N + 1}\Phi^{-1}(p_s) + \Phi^{-1}(1 - 2^{-\log_2 m}))^2}{S_N} p^{-1} \ .$$

In summary, this also shows that the data complexity of differential cryptanalysis is expected to be proportional to $1/p$. However, these results can only be used as a rough estimation, since the assumption that the counters $T_0, \ldots, T_{n-1}$ are independent is rather unrealistic.

From a theoretical perspective, the above approach gives the impression to be easily executable. From a practical point of view, though, an attacker has to overcome many obstacles, which were not mentioned above. Firstly, it is usually rather hard to find suitable differentials of a reasonably low probability over $r-1$ rounds for any decently designed cipher. It counts as a theoretical break if an attacker finds a differential whose probability p is larger than 2^{-b} where b is the block size (and often also the round key size) of the attacked cipher. For example, an attacker who targets a block cipher with 128-bit blocks (round keys) and who found an $r-1$-round differential having probability $p = 2^{-120}$, is able to mount an attack which can break the block cipher theoretically. From a practical perspective, though, the attack is infeasible, since gathering 2^{120} (or more) plaintext-ciphertext pairs is obviously impracticable. If p is sufficiently large so that the required amount of data could be indeed gathered then it counts also as a practical break. Secondly, differential attacks require plaintext pairs with a chosen difference which are harder to come by in practical scenarios than arbitrary plaintexts. Additionally, an attacker does not require only one or two of such pairs but usually a very large amount,

as has been pointed out already in the example above. Thirdly, the memory costs for keeping a large number of counters for a large number of plaintext-ciphertext pairs can be substantial.

Collisions. Mounting *collision attacks* against hash functions is a third use case of differentials. Suppose an attacker targets a hash function

$$\mathcal{H} : \mathbb{F}_2^n \times \mathbb{F}_2^m \to \mathbb{F}_2^n,\ (X, M) \mapsto \mathcal{H}(X, M)$$

where X is the *chaining value* [202] and M is a message block. Further assume he found a differential (α, β) of probability p in $\mathcal{H}$ where $\beta = 0$. The adversary processes message blocks M and $M \oplus \alpha$ and checks if $\mathcal{H}(X, M) \oplus \mathcal{H}(X, M \oplus \alpha) = 0$, i.e. if $\mathcal{H}(X, M) = \mathcal{H}(X, M \oplus \alpha)$ where X is the fixed IV of the hash function. On average, it is sufficient to process about $1/p$ message block pairs M and $M \oplus \alpha$ to find a collision. Differentials of the above type are usually called *vanishing differentials*. If the Hamming weight $\mathsf{hw}(\beta)$ of the output difference β is low, the differential can be used to produce so-called *near collisions* [56] which are often used as a first step to construct full collisions.

Prominent examples are the collision attacks against the hash functions SHA-0, SHA-1, and MD5 [239, 240, 241]. The attack against MD5 exploits non-trivial vanishing differentials and the one against SHA-0 basically starts from an impossible differential which is then modified to make it possible and construct collisions. Another speciality of the above MD5 attack is that it does not rely on XOR-differentials but instead uses differentials where differences are computed with respect to modular integer addition.

Further Techniques

Differential cryptanalysis actually covers not only the simple notions described in this part but includes a very broad range of techniques such as impossible differentials [54], boomerang attacks [238] or truncated and higher order differentials [159]. Although we briefly discuss impossible differentials in Chapter 5, the other techniques are not covered in this thesis and we refer the interested reader to the cited literature.

1.2.3 Linear Attacks

Although we present no new results on linear cryptanalysis [162] in this thesis, we nevertheless give a brief introduction to the basic concepts due to the general importance of the topic and for the sake of completeness.

Linear cryptanalysis was introduced by Matsui in 1992 to attack the block cipher FEAL [183] and was extended afterwards also to DES [182]. The latter belongs to the first publicly reported results on the cryptanalysis of DES. Linear cryptanalysis is, after differential cryptanalysis, the second most important technique for the analysis of cryptographic primitives. As already mentioned above, it has the advantage of requiring

only known plaintexts instead of chosen plaintexts as in the case of differential attacks. In general, however, linear attacks tend to be much less successful and much less versatile in comparison to their differential counterparts.

The basic idea of linear cryptanalysis is to build systems of equations which express key bits in terms of plaintext and ciphertext bits where non-linear components of the cipher are modelled by *linear approximations*. The attacker tries to construct such a linear approximation for the first $r-1$ rounds $f_{r-2} \circ \cdots \circ f_0$ of the cipher that holds with probability p such that $\alpha_0 \cdot x_0 \oplus \alpha_{r-1} \cdot x_{r-1} = 0$, where α_0 and α_{r-1} are so-called *linear masks* for the input and the output after $r-1$ rounds of the block cipher. We denote by $\cdot$ the *scalar product* over $\mathbb{F}_2$, i.e. given two elements $x = (x_0, \ldots, x_{n-1})$, $y = (y_0, \ldots, y_{n-1}) \in \mathbb{F}_2^n$, it is specified as $x \cdot y = \bigoplus_{i=0}^{n-1} x_i y_i$. The probability of the linear approximation is modelled as $p = 1/2 + \varepsilon$ where ε denotes the so-called *bias*. If $f_{r-2} \circ \cdots \circ f_0$ is an ideal block cipher then it is expected that $\varepsilon = 0$, i.e. the above equation holds with probability $1/2$. If the bias ε differs significantly from 0 and thus p deviates significantly from $1/2$, the linear approximation can be used to distinguish $f_{r-2} \circ \cdots \circ f_0$ from an ideal cipher by encrypting plaintexts x_0, chosen uniformly at random, to outputs x_{r-1} and checking how often the linear relation between them holds. Note that ε can be positive or negative but it is usually modelled as $0 \leq |\varepsilon| \leq \frac{1}{2}$ and the larger $|\varepsilon|$, the better it is for the attacker.

As in the case of differential attacks, once a distinguisher is found, it can be converted into a key recovery attack on the full block cipher $f_{r-1} \circ \cdots \circ f_0$ by guessing the bits of the last round subkey needed to determine the bits of x_{r-1} from the ciphertext x_r utilising the output mask α_{r-1}. The bias is estimated for each key candidate, and given enough plaintext-ciphertext pairs, the correct key candidate is likely to correspond to the one having the highest bias.

The notions of *linear probability*, *linear characteristics*, and *linear hulls* can be specified analogously to differential probability, differential characteristics and differentials [162]. An essential tool to compute the linear probability of a linear hull from its linear characteristics is the so-called *piling-up lemma* [162] which says that the probability p of the sum of m (independent) Boolean expressions holding with probabilities p_i, for $i \in \{0, \ldots, m-1\}$, can be computed by

$$p = \frac{1}{2} + 2^{m-1} \prod_{i=0}^{m-1} \left(p_i - \frac{1}{2} \right) \ .$$

An alternative and very frequently used tool in linear cryptanalysis for the study of Boolean functions is the representation of the latter through so-called *correlation matrices* [94]. In [220] not only the success probability of differential cryptanalysis with respect to the advantage over exhaustive search is investigated but also the one for linear attacks. Again, let p_s denote the success probability, let N denote the number of available plaintext-ciphertext pairs and suppose an adversary aims for an advantage of m bits over exhaustive

search. Then one gets

$$p_s = \Phi\left(2\varepsilon\sqrt{N} - \Phi^{-1}(1 - 2^{-m-1})\right)$$

under the assumption that the probability for a linear approximation is independent for each key candidate and is equal to $1/2$ for wrong guesses, and m and N being sufficiently large. This result can be reformulated again, in order to determine an estimation of the data requirement N depending on a given success probability p_s, as

$$N = \left(\frac{\Phi^{-1}(p_s) + \Phi^{-1}(1 - 2^{-m-1})}{2}\right)^2 \varepsilon^{-2} \ .$$

In other words, the data complexity is proportional to $1/\varepsilon^2$. In the same work it is furthermore verified experimentally that the above estimations for linear cryptanalysis are relatively precise, in contrast to the differential variant, which also indicates that the made assumptions seem to be quite realistic. Later Bogdanov and Tischhauser were able to derive a more accurate and better formula to determine the success probability p_s which, as a consequence, also improves the data complexity for linear attacks, see [75] for more information.

Although linear cryptanalysis is not as versatile as its differential counterpart, there exist several extensions for it. By considering multiple linear approximations in parallel, it has been shown that there are scenarios where the required amount of data can be reduced [148]. Knudsen showed for the case of DES that the complexity of linear attacks can be reduced through the exploitation of chosen-plaintexts [161]. Another generalisation of linear cryptanalysis by Knudsen uses non-linear approximations which can potentially lead to the recovery of additional information [163]. Moreover, the analogue technique to impossible differentials in linear cryptanalysis are the so-called zero-correlation attacks, proposed by Bogdanov and Rijmen in 2011, which exploit linear approximations having probabilities of exactly $1/2$, see [74, 76].

1.2.4 Algebraic Attacks

Attacking a cipher using algebraic cryptanalysis [24] usually consists of two steps: first, model the operations of the cipher algebraically by a system of non-linear multivariate polynomials $f_1, \ldots, f_s \in P$, where $P = K[x_1, \ldots, x_n]$ is a polynomial ring in n indeterminates $x_1, \ldots, x_n$ over a field K. In the case of symmetric cryptography, K is usually the finite field $\mathbb{F}_{2^r}$ of characteristic 2 with $r \geq 1$. Second, solve the resulting system of equations $\mathcal{S}$ given by

$$\begin{aligned} f_1(x_1, \ldots, x_n) &= 0 \\ &\vdots \\ f_s(x_1, \ldots, x_n) &= 0 \end{aligned}$$

for the key variables, thereby breaking the cipher. Although the basic idea sounds simple, the problem of solving generic systems of multivariate non-linear equations over finite fields is known to be NP-hard [117]. However, this is the generic complexity and there are also problem instances where solutions can be recovered efficiently. For example, the cipher Crypto-1 [91] was broken successfully using algebraic attacks. Moreover, algebraic techniques turn out to be quite efficient when used in combination with side-channel attacks [70] and are frequently used in the latter context.

Note that modelling a cipher through the polynomials $f_1, \ldots, f_s$ can be done in different ways and distinct representations might have a substantial impact on the solving time of the system [137]. In many cases, the polynomial system can be expressed purely as a function of the key indeterminates. However, the resulting polynomials usually have a huge degree and consist of a large number of monomials which is in general counter-productive for solving the system. In many cases, additional indeterminates and equations are therefore introduced. This results in an overdefined system, i.e. there are more equations than variables. Moreover, since the polynomial system might have solutions in the algebraic closure $\overline{\mathbb{F}}_{2^r}$ of $\mathbb{F}_{2^r}$, one usually adds the *field equations* $x_i^{2^r} - x_i = 0$ for $i \in \{1, \ldots, n\}$ to $\mathcal{S}$, which guarantees that all the solutions found by the solver are in $\mathbb{F}_{2^r}$. Solving such a system of equations can be done in various ways and we describe two approaches below.

Gröbner Bases

One classical approach to solve generic systems of non-linear multivariate polynomial equations are the so-called *Gröbner bases*, a tool from the field of computer algebra [169, 170]. A Gröbner basis is a generator set with special properties of an ideal $I = \langle f_1, \ldots, f_s \rangle \subset P$ and can be computed from the polynomials $f_1, \ldots, f_s$ that generate I. It allows to analyse many important properties of I and of its associated algebraic variety, i.e. the set of solutions of the equation system $f_1(x_1, \ldots, x_n) = 0, \ldots, f_s(x_1, \ldots, x_n) = 0$. For instance, Gröbner bases can be used to compute the dimension of ideals or to test for ideal equality or membership.

A Gröbner basis is specified with respect to a particular term ordering. For example, the *lexicographic term ordering* `lex` first compares the exponents of x_1 in the terms, and in case of equality compares exponents of x_2 and so on. The `lex` term ordering is very useful for applications like *elimination* [169], a technique that can be used to determine the solutions for a subset of the indeterminates appearing in a system of equations. However, computing a `lex` Gröbner basis is usually much harder than determining a Gröbner basis for certain other term orderings. A common approach is therefore to first calculate a non-`lex` Gröbner basis and convert it afterwards to a `lex` Gröbner basis using techniques like the *FGLM algorithm* [110].

The first algorithm to compute a Gröbner basis from an arbitrary ideal generator set was presented by Buchberger in his PhD thesis in 1965, and is nowadays better known

as *Buchberger's algorithm* [83]. It can be viewed as a generalisation of the Euclidean algorithm for univariate polynomials and of Gaussian elimination for linear polynomial systems. Later Faugère introduced the algorithms *F4* (1999) [108] and *F5* (2002) [109], two improved versions of Buchberger's algorithm which nowadays belong to the fastest known ways for computing Gröbner bases of generic systems of non-linear multivariate polynomial equations.

Modelling given problems through a system of polynomial equations is a common approach in many fields. Therefore, it is not surprising that Gröbner bases also found applications outside of computer algebra and algebraic geometry. For the special case of cryptography, equation systems often have a unique solution $(\alpha_1, \ldots, \alpha_n)$ in the base field K. Hence, the *reduced Gröbner basis* of the ideal I, i.e. a Gröbner basis satisfying certain minimality properties, is expected to have the form $\{x_1 - \alpha_1, \ldots, x_n - \alpha_n\}$.

SAT-Solvers

Boolean satisfiability (SAT) is another NP-complete decision problem targeting the question if a given Boolean formula is satisfiable. If it is indeed satisfiable, then a related question is how to find its solutions. Boolean formulas are constructed from a set of Boolean variables $\{x_1, \ldots, x_n\}$, the two constants `True` and `False`, and from the logical operations *conjunction* $\wedge$ (logical AND), *disjunction* $\vee$ (logical OR), and *negation* $\neg$ (logical NOT). An important set of problems involves finding assignments to the variables of a Boolean formula in *Conjunctive Normal Form* (CNF) such that the formula is `True`. These formulas are conjunctions of *clauses*, with a clause being a disjunction of *literals* and literals being either the variables x_i themselves or their negation $\neg x_i$. For example, $(x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3 \vee \neg x_4)$ is a Boolean formula in CNF.

SAT problems can be found in many different scientific fields, like artificial intelligence, bioinformatics, circuit design and test, automated theorem proving, and in cryptography. Due to their importance, there is a very high interest in solving such formulas in the most efficient way. Since SAT is NP-complete, the best algorithms for the generic case can only have an exponential worst-case run time as a lower bound. However, solving concrete instances can be done usually much faster than in exponential time, as it is often the case for NP-complete problems. Over the past years highly optimised SAT-solvers have been developed that are able to find solutions very efficiently and annual *SAT competitions* [217] continuously push the boundaries of solvable instances even further. Well-known solvers are *MiniSat* [198], *CryptoMiniSat* [181], and *Lingeling* (as well as its parallel versions *Plingeling* and *Treengeling*) [52]. All these solvers are based on the *conflict-driven clause learning algorithm* (CDCL) [26, 180], the modern variant of the *Davis-Putnam-Logeman-Loveland algorithm* (DPLL) [99] which explores the exponentially sized search space by a backtracking search procedure in order to find a satisfying variable assignment.

The power of these highly efficient SAT-solvers can be tapped for problems in crypt-

analysis as well. As usual, the first step is to model the target cipher algebraically by a system of polynomial equations. Afterwards the system is converted to a CNF [25, 137] which is then given as input to a SAT-solver. If a solution is found, it can be easily translated back to the algebraic world which then reveals the secret key of the cipher. In the special case of finite fields of characteristic 2, SAT-solvers are often the tool of choice due to their excellent performance. In case of more generic systems of equations over other fields (such as $\mathbb{Q}$) one usually falls back to more general solvers, like Gröbner bases. In Chapter 2, we give an example how SAT-solvers can be used for algebraic fault attacks, a combination of algebraic cryptanalyis and differential fault analysis.

An extension of SAT is the so-called *Satifiability Modulo Theories (SMT) problem* [100], a decision problem for logical formulas defined over first-order logic theories. Without going into the details, SMT instances allow to enrich Boolean formulas with additional theoretical concepts, such as integers, arrays or lists [115]. These extensions certainly remain NP-complete (and sometimes even become undecidable), but provide a much richer modelling language. As for SAT, there exist highly efficient SMT-solvers, like Boolector [82], STP [116], and Z3 [188], to name just a few, and annual competitions [218] which are a key ingredient for driving progress. SMT-solvers usually have a SAT-solver at their core which is then extended to handle the additional theoretical concepts. In Chapter 5, we are bridging the gap between algebraic and differential cryptanalysis and explore how to use SMT- and SAT-solvers to find differential characteristics.

1.2.5 Rotational Attacks

Rotational cryptanalysis was introduced by Khovratovich and Nikolić in [153] for the analysis of ARX-based cryptographic primitives. ARX refers to the operations (modular) integer addition, (bitwise) rotation, and `XOR` and is a quite common approach to design cryptographic algorithms. Prominent examples for such primitives are BLAKE(2) [16, 21], ChaCha [40], Salsa20 [41], SipHash [12], Skein [112], and Speck [27]. The idea of rotational cryptanalysis is to track propagation of rotational relations through a cryptographic transformation. Once rotation-invariant behaviour is detected, it can be used to construct distinguishers, mount key recovery attacks, and so forth, similarly as discussed for differential attacks. Rotational cryptanalysis was successfully applied to several simplified versions of cryptographic primitives including Skein [154] and Keccak [186].

In Chapter 5, we examine some rotational properties of the NORX core permutation.

1.2.6 Implementation Attacks

All attacks considered so far fall into the category of conventional cryptanalysis. In this section, we discuss *implementation* or *side-channel attacks* which are an extension of conventional cryptanalytical techniques. The adversarial models for conventional attacks as introduced above, provide the basis for implementation attacks as well, but additionally

an attacker is allowed to observe and sometimes even modify physical properties of the concrete implementation of an attacked algorithm. Additional data retrieved in this way can then be used in the subsequent (mathematical) cryptanalysis and usually drastically increases the chances for successful retrieval of secret information. This might even lead to scenarios where attackers are able to reconstruct secret information without having any knowledge on the processed plain- or ciphertexts at all. Implementation attacks obviously depend on the concrete realisation of the attacked algorithm in the physical world and therefore include a number of additional technical parameters, like implementation details, type of physical access, equipment of the attacker, number of algorithm executions, or number of required cryptographic devices.

Implementation attacks are categorised into *active* and *passive* versions. The latter derive secret information by observing operational parameters, such as execution time, power consumption, electromagnetic emissions or even cache behaviour. In contrast, active side-channel attacks actively interfere with the device, changing temporarily or even permanently parts of its state or execution flow, to retrieve additional information about the processed internal data. The so-called *fault-based attacks* belong to the field of active side-channel attacks and are particularly effective to break cryptographic systems, i.e. gain unauthorised access to the secret information. Implementation attacks are usually executed in three phases:

1. **Profiling Phase.** During profiling, an adversary has access to duplicates of the implementation that he plans to attack later. This might include, for example, hardware prototypes or simulation models. In this stage, the attacker has full control over all inputs (including secret keys) and outputs. The aim of this phase is to obtain information on the weaknesses of the implementation that can be exploited in subsequent steps when the real implementation with unknown secret information is targeted. Especially interesting are correlations between internal variables of the algorithm in the implementation and side-channel parameters that are leaked by the implementation.

2. **Online Phase.** In this stage, the attacker triggers the real implementation to execute the algorithm with some inputs and measures the physical side-channel parameters. Note that the secret information used by the implementation is unknown to the attacker in this phase. If the attacker is allowed to observe or choose inputs and outputs of the algorithm, which depends on the adversarial model, he can also record the data corresponding to these executions.

3. **Offline Phase.** During the offline phase, the attacker tries to recover the secret information (usually the unknown secret key) from the recorded side-channel parameters, the input and outputs, and from his knowledge of the implementation details obtained in the profiling phase.

Timing Analysis

Timing attacks exploit variations in the execution time of cryptographic algorithms. These variations often appear if there exist correlations between the running time of the primitive and secret input data. Gathering enough samples and analysing them with statistical methods, it is often possible to recover all of the secret information. The first side-channel attack exploiting timing analysis was proposed by Kocher [166] in 1996 where he observed timing leaks in implementations of public-key cryptosystems like Diffie-Hellman, RSA, and DSS. Later, researchers demonstrated practical network-based timing attacks against SSL-enabled web servers exploiting vulnerabilities in RSA implementations that relied on the *Chinese Remainder Theorem* [80, 81]. Although the actual network distance was small during their experiments, the attack successfully recovered the private key of the server within a couple of hours.

Kocher also noticed that queries to lookup tables in software can take varying amounts of time for different inputs due to RAM cache hits or misses. Lookup tables are frequently deployed in block ciphers in the form of non-linear S-boxes. If the inputs to such a lookup table depend on the secret key and enough timing samples can be gathered, then it is usually very easy to recover the key. These techniques were regularly exploited to mount attacks on various block ciphers including DES [234] and AES [35].

Timing attacks can be avoided by ensuring that all operations take the same amount of time regardless of the input. While this is often doable with some additional efforts, the performance of such implementations usually suffers significantly. Writing fast constant-time software is very often a highly complex task [149]. Thus, efforts increased in recent years to tackle this problem already on an algorithmic level by designing cryptographic primitives that have an inherited protection against timing attacks. These algorithms try to reduce the number of operations that might leak timing information and instead use building blocks that can be easily hardened against timing attacks or even better are executed in constant-time by default.

In Chapter 4, we introduce NORX, an authenticated encryption scheme, which was conceived according to the above approach, i.e. operations were carefully selected such that leakage of secret information through timing side-channels is prevented.

Power and Electromagnetic Analysis

Side-channel attacks which analyse the power consumption of devices implementing cryptographic algorithms were proposed by Kocher et al. [167] in 1999. They introduced two methods: *simple power analysis* (SPA) and *differential power analysis* (DPA).

SPA can yield information about the operation of a device as well as key material. It exploits the facts that different operations exhibit different power consumption behaviour and that these operations often depend on secret key information in a deterministic way. These dependencies are, in particular, mirrored in the power consumption profiles

of the respective operations. Therefore, it is sufficient for some ciphers to gather and visually examine a couple of power traces of the encryption process in order to obtain information on the secret key. For example, in the case of RSA it is often possible to distinguish between multiplication and squaring operations in exponentiation which then allows an adversary to reconstruct the secret key bit by bit [165]. While it is relatively easy to attack (unprotected) modular exponentiation (RSA) or scalar point multiplication (elliptic curve cryptosystems) with SPA, it is not necessarily as easy for other (symmetric) cryptographic primitives, such as block ciphers, due to their different internal structure. However, in the case of DES it was shown that SPA attacks targeting the key schedule allow an adversary to reconstruct certain parts of the secret key [165].

DPA involves statistical analysis of power consumption measurements and is generally more powerful than SPA. Advantages of DPA over SPA are that it allows to analyse power traces containing a high degree of noise and thus cannot be examined with SPA and that it is much more difficult to protect devices against DPA attacks. In a first step during DPA, an adversary guesses a part of the key. Using the guess and all known inputs (or outputs) he computes parts of the state. Afterwards all of the computed values are classified according to a leakage model and if the side-channel parameters exhibit correlations to the leakage model then the guess was correct. In the case of an unprotected AES-128 implementation it can be shown that at most 256 DPA traces are sufficient to recover one key byte. This means that at most $256 \cdot 16 = 4096$ DPA traces are required to recover the entire 128-bit key [165].

All of the above techniques can be used with measurements of electromagnetic emanations as well where the counterparts to SPA and DPA are *simple electromagnetic analysis* (SEMA) and *differential electromagnetic analysis* (DEMA) [1], respectively.

Fault-based Attacks

The effects of transient faults on electronic systems have been studied since the 1970s. At that time it was noticed that radioactive particles are capable of causing errors in chips by inducing small charges in sensitive chip areas which in turn could result in bit flips. One source for such faults are cosmic rays which are very weak at ground level due to the earth's atmosphere but can have more serious effects in the upper atmosphere or outer space. These observations led to further research aiming to better understand the effects of faults caused through environmental influences and to subsequently develop countermeasures preventing these kind of errors. A large driving force was the aerospace industry which was concerned about the effects such environmental-based faults might have on airborne electrical systems and especially on machines such as satellites.

The malicious exploitation of errors in electronic systems has been a more recent development. Fault-based attacks on cryptographic systems were introduced by Boneh et al. [77] in 1997. The researchers demonstrated that fault injections can be utilised to reveal the two secret prime numbers used in the RSA system thereby compromising the

security of the cipher. Over the years both symmetric and asymmetric cryptographic primitives have been attacked using fault-based cryptanalysis, including AES [191, 235], Trivium [130, 131], RSA [156], and elliptic curve cryptosystems [68]. As mentioned above, the three steps of fault attacks are:

Profiling Phase: Fault Analysis. In the first phase, the adversary performs an analysis of the cryptographic algorithm and the device he is trying to attack and determines those points during the execution of the encryption process which are suitable for a fault injection in the second phase. "Suitable" means here that the *fault propagation* behaviour after the injection satisfies certain conditions which ultimately lead to a successful reconstruction of secret data in the third step. Obviously, the techniques for the data analysis required in the final phase are also developed during this initial assessment. The requirements of the fault injection are usually summarised in a *fault model* and describe the temporal and spatial requirements of the fault injection.

Online Phase: Fault Injection and Data Collection. In the second phase, a physical disturbance (fault) is induced in the hardware on which the algorithm is executed. Some mechanisms to induce faults are summarised in [23] and include parasitic charge-carrier generation by a laser beam, manipulation of the circuit's clock, or reduction of the circuit's power-supply voltage. More recently proposed techniques use combinations of voltage and temperature based manipulations [173] or even insert highly stealthy hardware Trojans [172] into selected transistors of the cryptographic device. Those last two approaches especially excel at high-precision fault injections which are usually a fundamental requirement for being able to successfully mount a fault attack. More precisely, the success of a fault attack critically depends on the *spatial* and *temporal resolution* of the equipment of the attacker. Spatial resolution refers to the ability to accurately select the circuit element to be manipulated; temporal resolution stands for the capability to precisely determine the time (clock cycle) and the duration of fault injection. Several previously published attacks make different assumptions about vulnerable elements of the circuit accessible to the attacker and the required spatial and temporal resolutions [130, 156]. Most fault attacks are based on running the cryptographic algorithm several times in presence and in absence of the disturbance and gathering correct and faulty outputs.

Offline Phase: Evaluation and Reconstruction of Secret Data. In the third phase, the gathered data is used to derive the secret information using the methods developed in the first step. One particular effective approach to attack symmetric cryptographic algorithms is the so-called *differential fault analysis* [60] (DFA) which combines fault information with *differential cryptanalysis* [58]. It was employed for successful attacks on a variety of ciphers, including the initially mentioned attacks on AES [191, 235].

In Chapters 2 and 3, we study differential fault attacks in much more detail. We particularly investigate new techniques for DFA of the block ciphers LED, PRINCE and Bel-T and show how to successfully break them using realistic fault models. In case of LED-64, we additionally explore an algebraic variant of DFA.

1.3 Security Notions

In this section, we briefly recall some basic notions from provable security which are required later on in the thesis.

The notions of privacy (or confidentiality) and authenticity are central to the security of an AE(AD) scheme [32, 213, 214]. More specifically, an AE(AD) scheme is considered secure if and only if it is *IND-CPA* and *INT-CTXT*. The first notion, IND-CPA, refers to the property that ciphertexts produced by the AE(AD) scheme are indistinguishable from the random output of an ideal AE(AD) scheme. The second notion, INT-CTXT, requires that the AE(AD) scheme is secure against forgery attacks, i.e. that an adversary is not capable of producing a valid ciphertext-tag pair without knowledge of the secret key. These notions are usually specified in terms of security experiments. Below, we briefly recite the formal definitions and also explain their meanings in more detail.

Let $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be an AE(AD) scheme with key derivation function $\mathcal{K}$, encryption function $\mathcal{E}$ and decryption function $\mathcal{D}$ as specified in Section 1.1.5. Furthermore, let \$ denote the ideal version of $\mathcal{E}_K$ which returns a "ciphertext-tag" pair (C, T) chosen uniformly at random from $\mathbb{F}_2^{|P|+\mathsf{t}}$ on a query with (N, M) for an AE scheme or (N, A, M) for an AEAD scheme where N is a nonce, A a header, and M a message.

An adversary $\mathcal{A}$ is a probabilistic algorithm that has access to one or more oracles $\mathcal{O}$, denoted by $\mathcal{A}^{\mathcal{O}}$. We describe by $\mathcal{A}^{\mathcal{O}} \Rightarrow 1$ the event that $\mathcal{A}$, after interacting with an oracle $\mathcal{O}$, outputs 1. Adversaries have query access to encryption oracles of the underlying idealized primitives $\mathcal{E}$ or its counterpart \$ and possibly to decryption $\mathcal{D}$. We assume that an adversary $\mathcal{A}$ is *nonce-respecting* [213], which means he never makes two queries to $\mathcal{E}$ or \$ with the same nonce N for a fixed key K. However, $\mathcal{A}$ is allowed to repeat nonces for queries to $\mathcal{D}$. The key K is drawn uniformly at random from $\mathbb{F}_2^k$ at the beginning of a security experiment using $\mathcal{K}$.

IND-CPA Security. Consider the following security experiment: first, the oracle $\mathcal{O}$ tosses a fair coin to obtain a value $b \in \{0, 1\}$. Afterwards, the adversary $\mathcal{A}$ is allowed to send queries to $\mathcal{O}$. Depending on the value of b, $\mathcal{A}$ either obtains real encryptions from $\mathcal{E}_K$ or just random outputs from \$ for the messages he sends. The challenge for $\mathcal{A}$ is to guess b depending on the obtained answers from $\mathcal{O}$.

The *IND-CPA-advantage* of a computationally bounded adversary $\mathcal{A}$ who is trying to

break the privacy of Π is defined by

$$\mathbf{Adv}_{\Pi}^{\text{IND-CPA}}(\mathcal{A}) = \left|\Pr\left[K \xleftarrow{\$} \mathcal{K} : \mathcal{A}^{\mathcal{E}_K} \Rightarrow 1\right] - \Pr\left[\mathcal{A}^{\$} \Rightarrow 1\right]\right| .$$

We define $\mathbf{Adv}_{\Pi}^{\text{IND-CPA}}(t, q, l)$ to be the maximum advantage over all IND-CPA-adversaries $\mathcal{A}$ on Π that run in time at most t and make at most q queries of length at most l to the available oracles. The AE(AD) scheme Π is said to be *IND-CPA-secure* if the maximum advantage $\mathbf{Adv}_{\Pi}^{\text{IND-CPA}}(t, q, l)$ is negligible for any adversary $\mathcal{A}$ whose time-complexity is polynomial in the key size k.

INT-CTXT Security. Consider the following security experiment: Given access to two oracles, namely one for encryption $\mathcal{E}_K$ and one for decryption $\mathcal{D}_K$, the challenge for an adversary $\mathcal{A}$ is to generate a tuple (C, T) that has never been a response of $\mathcal{E}_K$ such that $\mathcal{D}_K$ on input of (N, C, T) for AE or of (N, A, C, T) for AEAD does not return $\perp$. If $\mathcal{A}$ is successful, we say he *forged* (C, T).

The *INT-CTXT-advantage* of a computationally bounded adversary $\mathcal{A}$ who is trying to break the authenticity of Π is defined by

$$\mathbf{Adv}_{\Pi}^{\text{INT-CTXT}}(\mathcal{A}) = \Pr\left[K \xleftarrow{\$} \mathcal{K} : \mathcal{A}^{\mathcal{E}_K, \mathcal{D}_K} \Rightarrow \text{forges}\right] .$$

We define $\mathbf{Adv}_{\Pi}^{\text{INT-CTXT}}(t, q, l)$ to be the maximum advantage over all INT-CTXT-adversaries $\mathcal{A}$ on Π that run in time at most t and make at most q queries of length at most l to the available oracles. The AE(AD) scheme Π is said to be *INT-CTXT-secure* if the maximum advantage $\mathbf{Adv}_{\Pi}^{\text{INT-CTXT}}(t, q, l)$ is negligible for any adversary $\mathcal{A}$ whose time-complexity is polynomial in the key size k.

Chapter 2

Fault-based Attacks on the Block Ciphers LED and PRINCE

2.1 Introduction

Attacking cryptographic primitives with conventional techniques is undoubtedly a necessary step in the evaluation of the security of any cryptographic construction. However, a much bigger threat usually originates from side-channel attacks targeted at concrete implementations of cryptographic algorithms and, in particular, unprotected implementations are usually an easy target. To secure such implementations, one first tries to uncover vulnerabilities of the unprotected versions and then constructs countermeasures which protect the implementations against the revealed attacks. In this chapter, we try to better understand the vulnerabilities of lightweight constructions which are often used to protect data in embedded devices, sensor networks or RFID chips, against implementation attacks. Especially, we investigate fault-based attacks on the LED and PRINCE, two lightweight block ciphers proposed for the usage in resource-constraint environments.

The fault attack on LED-64 was presented at COSADE 2012 [140] and its algebraic extension at SCC 2012 [138]. The multi-stage fault attacks on LED-128 and PRINCE [139] were part of the research presented at IOLTS 2014 [173] and FDTC 2014 [172].

Outline. The remainder of the chapter is organized as follows. The block ciphers LED and PRINCE are introduced in Sections 2.2 and 2.3. Next we present a detailed fault analysis of LED-64 in Section 2.4 and show that a single fault injection is sufficient to reconstruct the entire 64-bit secret key. Moreover, we motivate why a direct application of those techniques to LED-128 is not feasible. In Section 2.5, we then introduce the *multi-stage fault attack framework* which enables fault analysis of AES-like block ciphers having no or just a very lightweight key schedule, where subkeys are (almost) independent from each other. Furthermore, we show concrete applications of the framework to the block ciphers LED-128 and PRINCE and conclude that in both cases an average of 3 to 4 faults suffice to expose the 128-bit key. In Section 2.6, we discuss in detail how to extend the fault attack on LED-64 to an algebraic setting. First, we introduce the algebraic

model of LED using multivariate polynomials over $\mathbb{F}_2$ and then explain the integration of the information derived from the fault injection into the polynomial system. In the experimental phase, we present how the resulting problem description can be solved efficiently using SAT solvers. Finally, Section 2.7 concludes this chapter.

2.2 The Block Cipher LED

In this section, we recall the design of the block cipher LED, as specified in [126]. We first describe the general layout of LED in Section 2.2.1, followed by the mechanics of its round function in Section 2.2.2.

2.2.1 General Layout

As an AES-like block cipher, LED is based on a substitution-permutation network, i.e. its round function includes a substitution layer that transforms the words of the state through parallel application of S-boxes and a permutation layer consisting of a row permutation and a matrix multiplication which mixes the columns of the state. However, as a so-called lightweight block cipher, its field of application is focussed on environments with restricted resources. LED uses 64-bit blocks as states and accepts 64- and 128-bit keys. We denote these two versions by LED-64 and LED-128, respectively. Other key lengths, e.g. the popular choice of 80 bits, are padded to 128 bits by appending zeros until the desired key length is reached. Depending on the key size, the encryption algorithm performs 32 rounds for LED-64 and 48 round for LED-128. As already mentioned above, we discuss the components of such a round in the next Section. The 64-bit state S of LED is conceptually arranged in a 4×4 matrix

$$S = \begin{pmatrix} s_0 & s_1 & s_2 & s_3 \\ s_4 & s_5 & s_6 & s_7 \\ s_8 & s_9 & s_{10} & s_{11} \\ s_{12} & s_{13} & s_{14} & s_{15} \end{pmatrix}$$

where each of the 4-bit sized entry s_i, with $0 \leq i \leq 15$, is identified with an element of the finite field $\mathbb{F}_{16} \cong \mathbb{F}_2[x]/\langle x^4 + x + 1 \rangle$. In the following, we represent an element $g \in \mathbb{F}_{16}$, with $g = c_3x^3 + c_2x^2 + c_1x + c_0$ and $c_j \in \mathbb{F}_2$, by

$$g \mapsto c_3 \parallel c_2 \parallel c_1 \parallel c_0 \ .$$

In other words, this mapping identifies an element of $\mathbb{F}_{16}$ with a bit string. For example, the polynomial $x^3 + x + 1$ has the coefficient vector $(1, 0, 1, 1)$ and is mapped to the bit string `1011`. Note that we write 4-bit strings always in their hexadecimal short form, i.e. `1011` = `B`.

For encryption, a 64-bit plaintext unit M is considered as a 16-fold concatenation of 4-bit strings m_i with $0 \leq i \leq 15$, i.e. $M = m_0 \parallel m_1 \parallel \cdots \parallel m_{14} \parallel m_{15}$, which are identified with elements of $\mathbb{F}_{16}$ and arranged row-wise in a matrix of size 4×4:

$$M = \begin{pmatrix} m_0 & m_1 & m_2 & m_3 \\ m_4 & m_5 & m_6 & m_7 \\ m_8 & m_9 & m_{10} & m_{11} \\ m_{12} & m_{13} & m_{14} & m_{15} \end{pmatrix} .$$

Likewise, the key K is arranged in one or two matrices of size 4×4 over $\mathbb{F}_{16}$, depending on its size of either 64 bits or 128 bits:

- $|K| = 64$:

$$K = \begin{pmatrix} k_0 & k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 & k_7 \\ k_8 & k_9 & k_{10} & k_{11} \\ k_{12} & k_{13} & k_{14} & k_{15} \end{pmatrix} .$$

- $|K| = 128$:

$$K_1 = \begin{pmatrix} k_0 & k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 & k_7 \\ k_8 & k_9 & k_{10} & k_{11} \\ k_{12} & k_{13} & k_{14} & k_{15} \end{pmatrix} \quad K_2 = \begin{pmatrix} k_{16} & k_{17} & k_{18} & k_{19} \\ k_{20} & k_{21} & k_{22} & k_{23} \\ k_{24} & k_{25} & k_{26} & k_{27} \\ k_{28} & k_{29} & k_{30} & k_{31} \end{pmatrix} .$$

Figure 8 presents an overview of the encryption for both LED-64 and LED-128. Notice that key additions are performed only after four rounds have been executed. The authors of the original paper [126] call these four rounds a single Step. Keys are added by the function AddRoundKey (AK). It performs an addition of the state and key matrices using bitwise XOR. It is applied for input- and output-whitening as well as after every fourth round. Since LED-64 and LED-128 have 32 and 48 rounds, the ciphertext C is obtained after 8 and 12 Steps, respectively. Decryption is defined in the obvious way, by starting with a ciphertext C and going through all inverse operations in reverse order.

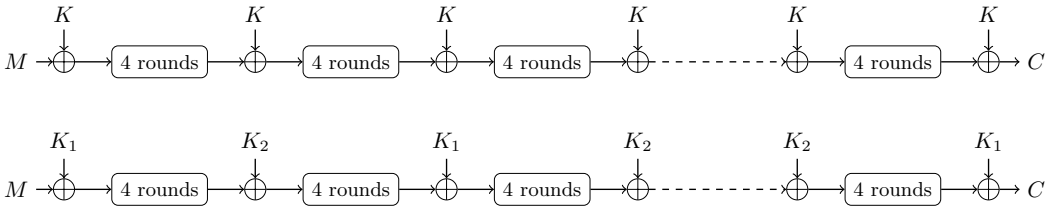


Figure 8: LED key usage: 64-bit key (top) and 128-bit key (bottom).

Figure 8 also exhibits a special feature of this cipher – there is no key schedule. On the one hand, this makes hardware implementations especially light-weight and on the other allows to derive simple security proofs even in a related-key attack model [126].

2.2.2 Round Function

Now we examine one round of the LED encryption algorithm. It is composed of the operations AddConstants (AC), SubCells (SC), ShiftRows (SR) and MixColumnsSerial (MCS). Figure 9 provides a rough overview. All matrices are defined over the field $\mathbb{F}_{16}$. Now we have a look at the individual steps.

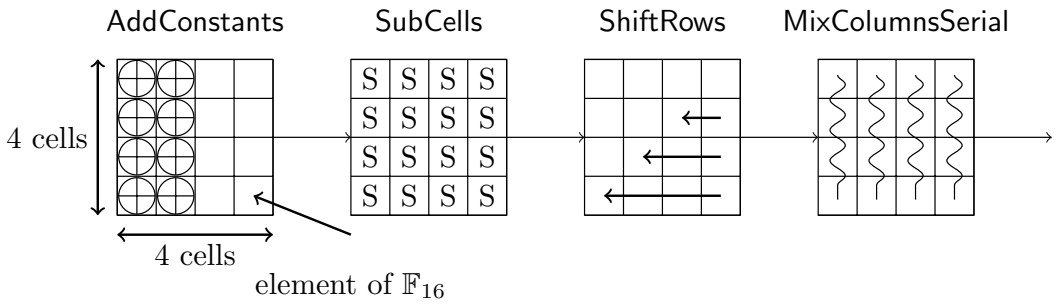


Figure 9: An overview of a single round of LED.

AddConstants (AC): For each round, a so-called *round constant* consisting of a tuple of six bits $(b_5, b_4, b_3, b_2, b_1, b_0)$ is defined as follows. First, the tuple is initialised to zero. In consecutive rounds, the starting point is the previous round constant. The six bits are shifted one position to the left and the new value of b_0 is computed as $b_5 + b_4 + 1$. This results in the round constants whose hexadecimal values are given in Table 1.

Table 1: The LED round constants.

Rounds	Constants
1 – 12	`01,03,07,0F,1F,3E,3D,3B,37,2F,1E,3C`
13 – 24	`39,33,27,0E,1D,3A,35,2B,16,2C,18,30`
25 – 36	`21,02,05,0B,17,2E,1C,38,31,23,06,0D`
37 – 48	`1B,36,2D,1A,34,29,12,24,08,11,22,04`

Next, the round constant is divided into $\mathtt{x} = b_5 \parallel b_4 \parallel b_3$ and $\mathtt{y} = b_2 \parallel b_1 \parallel b_0$ and $\mathtt{x}$ and $\mathtt{y}$ are interpreted as elements of $\mathbb{F}_{16}$. Moreover, let $(a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0)$ denote the eight bits encoding the key size of LED and let $\mathtt{u} = a_7 \parallel \ldots \parallel a_4$ and $\mathtt{v} = a_3 \parallel \ldots \parallel a_0$.

Finally, the matrix

$$\begin{pmatrix} 0 \oplus \mathtt{u} & \mathtt{x} & 0 & 0 \\ 1 \oplus \mathtt{u} & \mathtt{y} & 0 & 0 \\ 2 \oplus \mathtt{v} & \mathtt{x} & 0 & 0 \\ 3 \oplus \mathtt{v} & \mathtt{y} & 0 & 0 \end{pmatrix}$$

is formed and added to the state matrix using bitwise XOR.

SubCells (SC): Each entry x of the state matrix is replaced by the element $S[x]$ from the S-box given in Table 2. This particular S-box was first used by the block cipher PRESENT, see [71].

Table 2: The LED S-box.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S[x]$	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

ShiftRows (SR): For $i = 1, 2, 3, 4$, the ith row of the state matrix is shifted cyclically to the left by $i - 1$ positions:

$$\begin{pmatrix} s_0 & s_1 & s_2 & s_3 \\ s_4 & s_5 & s_6 & s_7 \\ s_8 & s_9 & s_{10} & s_{11} \\ s_{12} & s_{13} & s_{14} & s_{15} \end{pmatrix} \mapsto \begin{pmatrix} s_0 & s_1 & s_2 & s_3 \\ s_5 & s_6 & s_7 & s_4 \\ s_{10} & s_{11} & s_8 & s_9 \\ s_{15} & s_{12} & s_{13} & s_{14} \end{pmatrix} .$$

MixColumnsSerial (MCS): Each column v of the state matrix is replaced by the product $M \cdot v$, where M is the matrix[1]

$$M = \begin{pmatrix} 4 & 1 & 2 & 2 \\ 8 & 6 & 5 & 6 \\ \mathtt{B} & \mathtt{E} & \mathtt{A} & 9 \\ 2 & 2 & \mathtt{F} & \mathtt{B} \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 4 & 1 & 2 & 2 \end{pmatrix}^4 = (M')^4 .$$

Recall that multiplication of the elements is done over the finite field $\mathbb{F}_{16}$. M is a *Maximum Distance Separable* (MDS) matrix [145] and used for diffusion of the state. But there is another reason why M was chosen by the designers of LED: it can be decomposed into a product of four very simple and hardware-friendly matrices M', which in turn makes M suitable for compact serial implementations. This is obviously important for a lightweight block cipher targeted at running in resource-constrained environments.

[1] In the specification of LED in the original paper [126], the first row of M is given as 4 2 1 1. This appears to be a mistake, as the results computed starting with these values do not match those presented for the test vectors later in the paper. The matrix M used here is taken from the original authors' reference implementation of LED and gives the correct results for the test vectors.

2.3 The Block Cipher PRINCE

In this section, we recall the design of the block cipher PRINCE, as specified in [78]. We first describe the general layout of PRINCE in Section 2.3.1, followed by the mechanics of its round function in Section 2.3.2.

2.3.1 General Layout

PRINCE [78] is a 64-bit block cipher with a 128-bit secret key. Before an encryption (or decryption) is executed, the secret key $K = k_0 \parallel k_1$ is extended to 192 bit using the following mapping:

$$k_0 \parallel k_1 \mapsto k_0 \parallel k_1 \parallel k_2 := k_0 \parallel k_1 \parallel (k_0 \ggg 1) \oplus (k_0 \gg 63) \ .$$

The subkeys k_0 and k_2 are used for input- and output-whitening. The key k_1 is solely used in the core of PRINCE which is a 12-round block cipher. The described layout corresponds to the so-called FX-construction [62, 155]. Figure 10 gives an overview of the cipher.

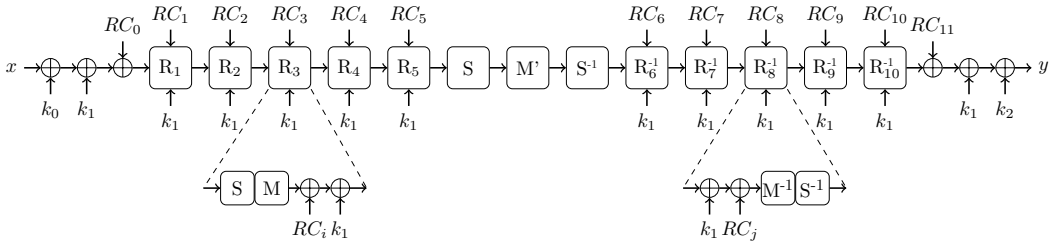


Figure 10: Layout of PRINCE.

Each round R_i and R_{i+5}^{-1} with $i \in \{1, \ldots, 5\}$ consists of a key addition, an S-layer, a linear layer which multiplies the state (represented through a 64-bit row vector) by a matrix, and the addition of a round constant RC_j, with $j \in \{0, \ldots, 11\}$. The rounds R_i and R_{i+5}^{-1} are separated by an involutive middle layer which consists of two S-layers, where S-boxes S and S^{-1} are applied, interleaved with the multiplication of the involutive matrix M'. The particular operations of a round are described in the next part.

2.3.2 Round Function

S-Layer: In this operation, PRINCE substitutes every element of the state using the 4-bit S-box depicted in Table 3.

M-/M'-Layer: The 64-bit state is multiplied by a 64×64 matrix M or M'. The matrix M' is only used in the middle-layer and thus has to be an involution to ensure

Table 3: The PRINCE S-box.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S[x]$	B	F	3	2	A	C	9	1	6	7	8	0	E	5	D	4

the α-reflection property (see below). For the specification of M' the following four bit matrices are introduced

$$M_0 = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad M_1 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$M_2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad M_3 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

which are arranged in two 16×16 matrices

$$\hat{M}_0 = \begin{pmatrix} M_0 & M_1 & M_2 & M_3 \\ M_1 & M_2 & M_3 & M_0 \\ M_2 & M_3 & M_0 & M_1 \\ M_3 & M_0 & M_1 & M_2 \end{pmatrix} \qquad \hat{M}_1 = \begin{pmatrix} M_1 & M_2 & M_3 & M_0 \\ M_2 & M_3 & M_0 & M_1 \\ M_3 & M_0 & M_1 & M_2 \\ M_0 & M_1 & M_2 & M_3 \end{pmatrix} .$$

Both matrices are involutions due to their symmetric structure and the choice of the building blocks $M_0, \ldots, M_3$. Finally, M' is specified as the following block diagonal matrix

$$M' = \begin{pmatrix} \hat{M}_0 & 0 & 0 & 0 \\ 0 & \hat{M}_1 & 0 & 0 \\ 0 & 0 & \hat{M}_1 & 0 \\ 0 & 0 & 0 & \hat{M}_0 \end{pmatrix}$$

which is a permutation on 64-bit strings. The matrix M' is used to specify the second matrix M, via $M = SR \circ M'$, where SR corresponds to the ShiftRows operation as used in AES, which permutes the sixteen 4-bit elements of a 64-bit string by

$$(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15) \mapsto (0, 5, 10, 15, 4, 9, 14, 3, 8, 13, 2, 7, 12, 1, 6, 11) .$$

Note that multiplication with M is not an involution anymore, since ShiftRows is non-involutive.

RC_i-add: This operation adds the round constant RC_i to the state using bitwise XOR. The values of RC_i are defined in Table 4. Those constants have the special property that, for all $i \in \{1, \ldots, 10\}$, the equality

$$RC_i \oplus RC_{11-i} = \texttt{C0AC29B7C97C50DD} \ (=: \alpha)$$

holds. Together with the fact that M' is involutive, it is possible to perform encryption and decryption using basically the same circuit (or implementation) and is referred to as the α-*reflection property*. For more details, see the specification of PRINCE [78].

Table 4: The PRINCE round constants.

i	RC_i
0 – 2	0000000000000000, 13198A2E03707344, A4093822299F31D0,
3 – 5	082EFA98EC4E6C89, 452821E638D01377, BE5466CF34E90C6C,
6 – 8	7EF84F78FD955CB1, 85840851F1AC43AA, C882D32F25323C54,
9 – 11	64A51195E0E3610D, D3B5A399CA0C2399, C0AC29B7C97C50DD

2.4 Fault Attacks on LED-64

In this part, we describe how to cryptanalyse LED-64, the 64-bit version of the LED block cipher, by fault injection followed by differential fault analysis. Our fault model assumes that an attacker is capable of inducing a fault in a particular 4-bit entry of the state matrix at a specified point during the encryption algorithm, changing it to a random and unknown value. The attack is based on solving *fault equations* derived from the propagation of this fault through the remainder of the encryption algorithm. We start with an introduction to the fault models used throughout the rest of the chapter. Then we present a general outline of the fault attack on LED-64, show how to construct the fault equations and discuss their usage for key filtering. Finally, we illustrate our experimental results and present a first idea how to extend the attack to LED-128.

2.4.1 Fault Models

The assumptions on the fault-injection capabilities of an attacker must be formalised in a *fault model*. In this part, we employ two of them, namely

- the *random and known fault model* (RKF) and
- the *random and unknown fault model* (RUF).

Our focus rests on block ciphers, which have an n-bit sized state and whose operations usually work on m-bit sized parts of the latter. We therefore assume that a fault f perturbs exactly one of those m-bit sized elements and leaves the other $n - m$ bits unaffected. In case of LED (and also later for PRINCE) we have $m = 4$. In other words, a fault injection affects precisely one nibble of the state. We represent f as a bit string with values 1 on the positions where the state is flipped, i.e., the fault injection is described by a bitwise XOR of f to the state. Consequently, there are at most $n/m \cdot (2^m - 1)$ different faults for a given point of time during encryption which amounts to 240 in case of LED and PRINCE. The RKF model assumes that the attacker can target a specific nibble, e.g., the very first one that includes state bit positions 0 through $m - 1$. The RUF model assumes that the fault injection perturbs a randomly selected nibble and that it cannot be observed which nibble was affected. The RUF model is weaker and therefore easier to match by practical fault-injection equipment, but it requires more complex mathematical analysis.

2.4.2 Fault Equations

Fault Propagation

The attack starts with a fault injection at the beginning of round $r = 30$. The attacker then watches the error spread over the state matrix in the course of the last three rounds. Figure 11 shows the propagation of a fault injected in the first entry of the state matrix during the encryption. Every square depicts the XOR difference of the correct and the faulty cipher state during that particular phase of the last three encryption rounds.

Afterwards, the attacker querys the encryption of the plaintext again, but this time without a fault injection and ends up with two ciphertexts, the correct $C = c_0 \parallel \ldots \parallel c_{15}$ and the faulty $C' = c'_0 \parallel \ldots \parallel c'_{15}$, with c_i, $c'_i \in \mathbb{F}_{16}$ for $0 \leq i \leq 15$. By working backwards from these results, we construct equations that describe relations between C and C'. Such relations exist, because the difference between C and C' is due to a single faulty state matrix entry at the beginning of round 30.

With the help of those equations we then try to limit the space of all possible keys, such that we are able to perform a brute force attack, or in the best case, get the secret key directly. Next, we discuss the method to establish the fault equations.

Inversion of LED Steps

We consider C resp. C' as a starting point and invert every operation of the encryption until the beginning of round $r = 30$. The 4-bit sized elements k_i with $0 \leq i \leq 15$ of the key are viewed as indeterminates. The following steps list the expressions one has to compute to finally get the fault equations.

1. AK^{-1}: $c_i + k_i$ and $c'_i + k_i$.

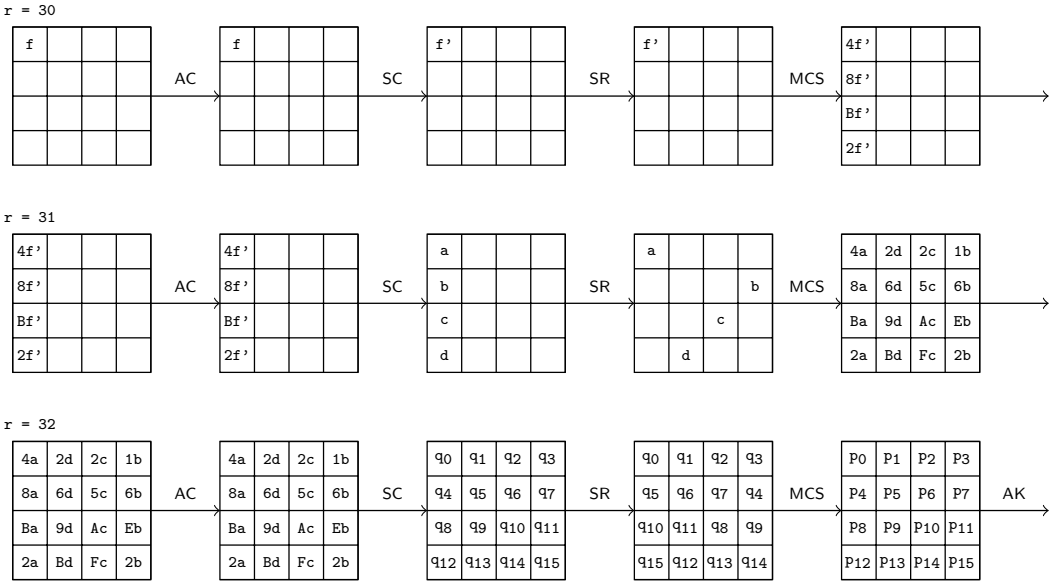


Figure 11: Fault propagation in the LED cipher.

2. MCS^{-1}: Use the inverse matrix

$$M^{-1} = \begin{pmatrix} \mathtt{C} & \mathtt{C} & \mathtt{D} & \mathtt{4} \\ \mathtt{3} & \mathtt{8} & \mathtt{4} & \mathtt{5} \\ \mathtt{7} & \mathtt{6} & \mathtt{2} & \mathtt{E} \\ \mathtt{D} & \mathtt{9} & \mathtt{9} & \mathtt{D} \end{pmatrix}$$

of the matrix M from the MCS operation to get the expressions

$$\mathtt{C} \cdot (c_0 + k_0) + \mathtt{C} \cdot (c_4 + k_4) + \mathtt{D} \cdot (c_8 + k_8) + \mathtt{4} \cdot (c_{12} + k_{12}) \text{ resp.}$$
$$\mathtt{C} \cdot (c'_0 + k_0) + \mathtt{C} \cdot (c'_4 + k_4) + \mathtt{D} \cdot (c'_8 + k_8) + \mathtt{4} \cdot (c'_{12} + k_{12}) \ .$$

Obviously, the other expressions are computed in a similar way.

3. SR^{-1}: As the operation only shifts the entries of the state matrix, the computed expressions are unaffected.

4. SC^{-1}: Inverting the SC operation results in

$$S^{-1}(\mathtt{C} \cdot (c_0 + k_0) + \mathtt{C} \cdot (c_4 + k_4) + \mathtt{D} \cdot (c_8 + k_8) + \mathtt{4} \cdot (c_{12} + k_{12})) \text{ resp.}$$
$$S^{-1}(\mathtt{C} \cdot (c'_0 + k_0) + \mathtt{C} \cdot (c'_4 + k_4) + \mathtt{D} \cdot (c'_8 + k_8) + \mathtt{4} \cdot (c'_{12} + k_{12}))$$

where S^{-1} is the inverse of the LED S-box, as shown in Table 5. The remaining expressions are computed in the same way again.

Table 5: The inverse LED S-box.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S^{-1}(x)$	5	E	F	8	C	1	2	D	B	4	6	3	0	7	9	A

Generation of Fault Equations

The XOR difference between the two related expressions, one derived from C and the other one from C', is computed and identified with the corresponding (unknown) fault value. In case of a fault injected into the first state element s_0, the fault value can be read off the fault propagation in Figure 11 above. In this case, we get

$$\begin{aligned} \mathtt{4} \cdot a = S^{-1}(\mathtt{C} \cdot (c_0 + k_0) + \mathtt{C} \cdot (c_4 + k_4) + \mathtt{D} \cdot (c_8 + k_8) + \mathtt{4} \cdot (c_{12} + k_{12})) + \\ S^{-1}(\mathtt{C} \cdot (c'_0 + k_0) + \mathtt{C} \cdot (c'_4 + k_4) + \mathtt{D} \cdot (c'_8 + k_8) + \mathtt{4} \cdot (c'_{12} + k_{12})) \ . \end{aligned}$$

In summary, we get 16 fault equations for a fault injected at a particular 4-bit element of the state matrix at the beginning of round $r = 30$. Before we list the fault equations, we pause to introduce some notation. Let a, b, c, and d be indeterminates. Their concrete values are determined in the evaluation phase of the attack and depend on the injected fault value. In contrast to the example above, faults can occur in each of the 16 state elements in round $r = 30$ and hence we need a more general approach to be able to model all possibilities. Let $l \in \{0, \ldots, 15\}$ denote the fault location and let $e_0, \ldots, e_{15}$ denote indeterminates, whose values depend on l. We call $e_0, \ldots, e_{15}$ the *fault coefficients*. Analysing the fault location l in a similar way as in Figure 11, we end up with four categories, and thus four configurations for $e_0, \ldots, e_{15}$. The possible assignments are shown in Figure 12.

$$(e_0, \ldots, e_{15}) = \begin{cases} (\mathtt{4}, \mathtt{2}, \mathtt{2}, \mathtt{1}, \mathtt{8}, \mathtt{6}, \mathtt{5}, \mathtt{6}, \mathtt{B}, \mathtt{9}, \mathtt{A}, \mathtt{E}, \mathtt{2}, \mathtt{B}, \mathtt{F}, \mathtt{2}), & \text{if } l \in \{0, 5, 10, 15\} \\ (\mathtt{1}, \mathtt{4}, \mathtt{2}, \mathtt{2}, \mathtt{6}, \mathtt{8}, \mathtt{6}, \mathtt{5}, \mathtt{E}, \mathtt{B}, \mathtt{9}, \mathtt{A}, \mathtt{2}, \mathtt{2}, \mathtt{B}, \mathtt{F}), & \text{if } l \in \{1, 6, 11, 12\} \\ (\mathtt{2}, \mathtt{1}, \mathtt{4}, \mathtt{2}, \mathtt{5}, \mathtt{6}, \mathtt{8}, \mathtt{6}, \mathtt{A}, \mathtt{E}, \mathtt{B}, \mathtt{9}, \mathtt{F}, \mathtt{2}, \mathtt{2}, \mathtt{B}), & \text{if } l \in \{2, 7, 8, 13\} \\ (\mathtt{2}, \mathtt{2}, \mathtt{1}, \mathtt{4}, \mathtt{6}, \mathtt{5}, \mathtt{6}, \mathtt{8}, \mathtt{9}, \mathtt{A}, \mathtt{E}, \mathtt{B}, \mathtt{B}, \mathtt{F}, \mathtt{2}, \mathtt{2}), & \text{if } l \in \{3, 4, 9, 14\} \end{cases}$$

Figure 12: Fault coefficients for the LED key tuple filtering equations.

The fault equations are denoted by $E_{x,i}$, where $x \in \{a, b, c, d\}$ identifies the block the equation belongs to and $i \in \{0, 1, 2, 3\}$ the number of the equation. Figure 13 lists the 16 equations in their most general form. For a concrete instance of the attack, we assume that we are given the correct ciphertext C and the faulty ciphertext C' and we

assume henceforth that these values have been substituted for the variables c_j and c'_j, for $0 \leq j \leq 15$, in the fault equations.

2.4.3 Key Filtering Mechanisms

The correct key satisfies all the fault equations derived in Figure 13. Our attack is based on quickly identifying large sets of key candidates which are inconsistent with some of the fault equations and excluding these sets from further consideration. The attack stops when the number of remaining key candidates is so small that exhaustive search becomes feasible. Key candidates are organized using a formalism called *fault tuples* (introduced below), and filters work directly on fault tuples. The outline of our approach is as follows:

1. **Key Tuple Filtering.** Filter the key tuples and obtain the fault tuples together with their key candidate sets. (This stage is partially inspired by the evaluation of the fault equations in [191] and [235]).

2. **Key Set Filtering.** Filter the fault tuples to eliminate wrong key candidate sets.

3. **Exhaustive Search.** Find the correct key by considering all key candidates left over from filtering.

Details on the individual stages and the parameter choice for the attacks are given below.

Key Tuple Filtering

In the following, let $x \in \{a, b, c, d\}$ and $i \in \{1, 2, 3, 4\}$. Each equation $E_{x,i}$ depends only on four key indeterminates, see Figure 13. In the first stage, we start by computing a list $S_{x,i}$ of length 16 for each equation $E_{x,i}$. The jth entry of $S_{x,i}$, denoted $S_{x,i}(j)$, is the set of all 4-tuples of values of key indeterminates which produces the jth field element as a result of evaluating equation $E_{x,i}$ at these values. Notice that we have to check 16^4 tuples of elements of $\mathbb{F}_{16}$ in order to generate one $S_{x,i}(j)$. The computation of all entries $S_{x,i}(j)$ requires merely 16^5 evaluations of simple polynomials over $\mathbb{F}_{16}$. Since all entries are independent from each other, the calculations can be performed in parallel.

In the next step, we determine, for every $x \in \{a, b, c, d\}$ the set of possible values j_x of x such that $S_{x,0}(j_x)$, $S_{x,1}(j_x)$, $S_{x,2}(j_x)$, and $S_{x,3}(j_x)$ are all non-empty. In other words, we are looking for j_x which can occur on the left-hand side of equations $E_{x,0}$, $E_{x,1}$, $E_{x,2}$, and $E_{x,3}$ for some possible values of key indeterminates. We call an identified value $j_x \in \mathbb{F}_{16}$ a *possible fault value* of x.

By combining the possible fault values of a, b, c, d in all available ways, we obtain tuples $t = (j_a, j_b, j_c, j_d)$ which we call *fault tuples* of the given pair (C, C'). For each fault tuple,

$$
\begin{aligned}
e_0 \cdot a &= S^{-1}(\mathtt{C} \cdot (c_0 + k_0) + \mathtt{C} \cdot (c_4 + k_4) + \mathtt{D} \cdot (c_8 + k_8) + \mathtt{4} \cdot (c_{12} + k_{12})) + \\
&\quad S^{-1}(\mathtt{C} \cdot (c'_0 + k_0) + \mathtt{C} \cdot (c'_4 + k_4) + \mathtt{D} \cdot (c'_8 + k_8) + \mathtt{4} \cdot (c'_{12} + k_{12})) && (E_{a,0}) \\
e_4 \cdot a &= S^{-1}(3 \cdot (c_3 + k_3) + \mathtt{8} \cdot (c_7 + k_7) + \mathtt{4} \cdot (c_{11} + k_{11}) + \mathtt{5} \cdot (c_{15} + k_{15})) + \\
&\quad S^{-1}(3 \cdot (c'_3 + k_3) + \mathtt{8} \cdot (c'_7 + k_7) + \mathtt{4} \cdot (c'_{11} + k_{11}) + \mathtt{5} \cdot (c'_{15} + k_{15})) && (E_{a,1}) \\
e_8 \cdot a &= S^{-1}(7 \cdot (c_2 + k_2) + \mathtt{6} \cdot (c_6 + k_6) + \mathtt{2} \cdot (c_{10} + k_{10}) + \mathtt{E} \cdot (c_{14} + k_{14})) + \\
&\quad S^{-1}(7 \cdot (c'_2 + k_2) + \mathtt{6} \cdot (c'_6 + k_6) + \mathtt{2} \cdot (c'_{10} + k_{10}) + \mathtt{E} \cdot (c'_{14} + k_{14})) && (E_{a,2}) \\
e_{12} \cdot a &= S^{-1}(\mathtt{D} \cdot (c_1 + k_1) + \mathtt{9} \cdot (c_5 + k_5) + \mathtt{9} \cdot (c_9 + k_9) + \mathtt{D} \cdot (c_{13} + k_{13})) + \\
&\quad S^{-1}(\mathtt{D} \cdot (c'_1 + k_1) + \mathtt{9} \cdot (c'_5 + k_5) + \mathtt{9} \cdot (c'_9 + k_9) + \mathtt{D} \cdot (c'_{13} + k_{13})) && (E_{a,3}) \\
\\
e_3 \cdot b &= S^{-1}(\mathtt{C} \cdot (c_3 + k_3) + \mathtt{C} \cdot (c_7 + k_7) + \mathtt{D} \cdot (c_{11} + k_{11}) + \mathtt{4} \cdot (c_{15} + k_{15})) + \\
&\quad S^{-1}(\mathtt{C} \cdot (c'_3 + k_3) + \mathtt{C} \cdot (c'_7 + k_7) + \mathtt{D} \cdot (c'_{11} + k_{11}) + \mathtt{4} \cdot (c'_{15} + k_{15})) && (E_{b,0}) \\
e_7 \cdot b &= S^{-1}(3 \cdot (c_2 + k_2) + \mathtt{8} \cdot (c_6 + k_6) + \mathtt{4} \cdot (c_{10} + k_{10}) + \mathtt{5} \cdot (c_{14} + k_{14})) + \\
&\quad S^{-1}(3 \cdot (c'_2 + k_2) + \mathtt{8} \cdot (c'_6 + k_6) + \mathtt{4} \cdot (c'_{10} + k_{10}) + \mathtt{5} \cdot (c'_{14} + k_{14})) && (E_{b,1}) \\
e_{11} \cdot b &= S^{-1}(7 \cdot (c_1 + k_1) + \mathtt{6} \cdot (c_5 + k_5) + \mathtt{2} \cdot (c_9 + k_9) + \mathtt{E} \cdot (c_{13} + k_{13})) + \\
&\quad S^{-1}(7 \cdot (c'_1 + k_1) + \mathtt{6} \cdot (c'_5 + k_5) + \mathtt{2} \cdot (c'_9 + k_9) + \mathtt{E} \cdot (c'_{13} + k_{13})) && (E_{b,2}) \\
e_{15} \cdot b &= S^{-1}(\mathtt{D} \cdot (c_0 + k_0) + \mathtt{9} \cdot (c_4 + k_4) + \mathtt{9} \cdot (c_8 + k_8) + \mathtt{D} \cdot (c_{12} + k_{12})) + \\
&\quad S^{-1}(\mathtt{D} \cdot (c'_0 + k_0) + \mathtt{9} \cdot (c'_4 + k_4) + \mathtt{9} \cdot (c'_8 + k_8) + \mathtt{D} \cdot (c'_{12} + k_{12})) && (E_{b,3}) \\
\\
e_2 \cdot c &= S^{-1}(\mathtt{C} \cdot (c_2 + k_2) + \mathtt{C} \cdot (c_6 + k_6) + \mathtt{D} \cdot (c_{10} + k_{10}) + \mathtt{4} \cdot (c_{14} + k_{14})) + \\
&\quad S^{-1}(\mathtt{C} \cdot (c'_2 + k_2) + \mathtt{C} \cdot (c'_6 + k_6) + \mathtt{D} \cdot (c'_{10} + k_{10}) + \mathtt{4} \cdot (c'_{14} + k_{14})) && (E_{c,0}) \\
e_6 \cdot c &= S^{-1}(3 \cdot (c_1 + k_1) + \mathtt{8} \cdot (c_5 + k_5) + \mathtt{4} \cdot (c_9 + k_9) + \mathtt{5} \cdot (c_{13} + k_{13})) + \\
&\quad S^{-1}(3 \cdot (c'_1 + k_1) + \mathtt{8} \cdot (c'_5 + k_5) + \mathtt{4} \cdot (c'_9 + k_9) + \mathtt{5} \cdot (c'_{13} + k_{13})) && (E_{c,1}) \\
e_{10} \cdot c &= S^{-1}(7 \cdot (c_0 + k_0) + \mathtt{6} \cdot (c_4 + k_4) + \mathtt{2} \cdot (c_8 + k_8) + \mathtt{E} \cdot (c_{12} + k_{12})) + \\
&\quad S^{-1}(7 \cdot (c'_0 + k_0) + \mathtt{6} \cdot (c'_4 + k_4) + \mathtt{2} \cdot (c'_8 + k_8) + \mathtt{E} \cdot (c'_{12} + k_{12})) && (E_{c,2}) \\
e_{14} \cdot c &= S^{-1}(\mathtt{D} \cdot (c_3 + k_3) + \mathtt{9} \cdot (c_7 + k_7) + \mathtt{9} \cdot (c_{11} + k_{11}) + \mathtt{D} \cdot (c_{15} + k_{15})) + \\
&\quad S^{-1}(\mathtt{D} \cdot (c'_3 + k_3) + \mathtt{9} \cdot (c'_7 + k_7) + \mathtt{9} \cdot (c'_{11} + k_{11}) + \mathtt{D} \cdot (c'_{15} + k_{15})) && (E_{c,3}) \\
\\
e_1 \cdot d &= S^{-1}(\mathtt{C} \cdot (c_1 + k_1) + \mathtt{C} \cdot (c_5 + k_5) + \mathtt{D} \cdot (c_9 + k_9) + \mathtt{4} \cdot (c_{13} + k_{13})) + \\
&\quad S^{-1}(\mathtt{C} \cdot (c'_1 + k_1) + \mathtt{C} \cdot (c'_5 + k_5) + \mathtt{D} \cdot (c'_9 + k_9) + \mathtt{4} \cdot (c'_{13} + k_{13})) && (E_{d,0}) \\
e_5 \cdot d &= S^{-1}(3 \cdot (c_0 + k_0) + \mathtt{8} \cdot (c_4 + k_4) + \mathtt{4} \cdot (c_8 + k_8) + \mathtt{5} \cdot (c_{12} + k_{12})) + \\
&\quad S^{-1}(3 \cdot (c'_0 + k_0) + \mathtt{8} \cdot (c'_4 + k_4) + \mathtt{4} \cdot (c'_8 + k_8) + \mathtt{5} \cdot (c'_{12} + k_{12})) && (E_{d,1}) \\
e_9 \cdot d &= S^{-1}(7 \cdot (c_3 + k_3) + \mathtt{6} \cdot (c_7 + k_7) + \mathtt{2} \cdot (c_{11} + k_{11}) + \mathtt{E} \cdot (c_{15} + k_{15})) + \\
&\quad S^{-1}(7 \cdot (c'_3 + k_3) + \mathtt{6} \cdot (c'_7 + k_7) + \mathtt{2} \cdot (c'_{11} + k_{11}) + \mathtt{E} \cdot (c'_{15} + k_{15})) && (E_{d,2}) \\
e_{13} \cdot d &= S^{-1}(\mathtt{D} \cdot (c_2 + k_2) + \mathtt{9} \cdot (c_6 + k_6) + \mathtt{9} \cdot (c_{10} + k_{10}) + \mathtt{D} \cdot (c_{14} + k_{14})) + \\
&\quad S^{-1}(\mathtt{D} \cdot (c'_2 + k_2) + \mathtt{9} \cdot (c'_6 + k_6) + \mathtt{9} \cdot (c'_{10} + k_{10}) + \mathtt{D} \cdot (c'_{14} + k_{14})) && (E_{d,3})
\end{aligned}
$$

Figure 13: The LED fault equations for key tuple filtering.

we intersect those sets $S_{x,i}(j_x)$ which correspond to equations involving the same key indeterminates:

$$\begin{aligned}(k_0, k_4, k_8, k_{12}) &: S_{a,0}(j_a) \cap S_{b,3}(j_b) \cap S_{c,2}(j_c) \cap S_{d,1}(j_d)\\ (k_1, k_5, k_9, k_{13}) &: S_{a,3}(j_a) \cap S_{b,2}(j_b) \cap S_{c,1}(j_c) \cap S_{d,0}(j_d)\\ (k_2, k_6, k_{10}, k_{14}) &: S_{a,2}(j_a) \cap S_{b,1}(j_b) \cap S_{c,0}(j_c) \cap S_{d,3}(j_d)\\ (k_3, k_7, k_{11}, k_{15}) &: S_{a,1}(j_a) \cap S_{b,0}(j_b) \cap S_{c,3}(j_c) \cap S_{d,2}(j_d)\ .\end{aligned}$$

By recombining the key values $(k_0, \ldots, k_{15})$ using all possible choices in these four intersections, we arrive at the *key candidate set* for the given fault tuple. If the size of the key candidate sets is sufficiently small, it is possible to skip the second stage of the attack and search all key candidate sets exhaustively for the correct key.

Each of the above intersections contains typically $2^4 - 2^8$ elements. Consequently, we expect key candidate sets with $2^{16} - 2^{32}$ elements for one fault tuple. Unfortunately, often several fault tuples are generated. The key candidate sets corresponding to different fault tuples are pairwise disjoint by construction. Only one of them contains the true secret key, but up to now we lack a way to distinguish the correct key candidate set (i.e. the one containing the true key) from the wrong ones. Therefore, we would need to search through all generated key candidate sets in the final bruteforce. Before we address this problem in the next section, we illustrate key set filtering by an example.

We take one of the official test vectors from the LED specification and apply our attack. It is given by

$$\begin{aligned}K &= \texttt{01234567 89ABCDEF}\\ M &= \texttt{01234567 89ABCDEF}\\ C &= \texttt{A003551E 3893FC58}\\ C' &= \texttt{DBBA6F7B 1DED088C}\end{aligned}$$

where the faulty ciphertext C' is obtained when injecting the error $e = \texttt{3}$ in the first entry of the state matrix at the beginning of the 30th round. Although the attack is independent of the value of the error, we use a specific one here in order to enable the reader to reproduce our results. Evaluation of the fault equations provides us with the following tables:

a	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$\#S_{a,0}$	0	2^{14}	0	2^{14}	0	0	0	0	0	$\mathbf{2^{14}}$	0	2^{14}	0	0	0	0
$\#S_{a,1}$	0	0	0	0	2^{13}	2^{13}	2^{14}	0	0	$\mathbf{2^{13}}$	0	2^{13}	0	2^{13}	2^{13}	0
$\#S_{a,2}$	0	0	0	2^{13}	0	2^{14}	0	2^{13}	0	$\mathbf{2^{13}}$	0	0	2^{13}	0	2^{13}	2^{13}
$\#S_{a,3}$	0	0	2^{13}	0	2^{13}	0	2^{13}	2^{13}	0	$\mathbf{2^{14}}$	0	2^{13}	2^{13}	0	0	0

b	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$\#S_{b,0}$	0	0	2^{13}	2^{14}	$\mathbf{2^{13}}$	0	2^{13}	2^{13}	0	2^{13}	0	0	0	0	2^{13}	0
$\#S_{b,1}$	0	2^{13}	2^{13}	2^{13}	$\mathbf{2^{14}}$	0	2^{13}	0	0	0	2^{13}	0	2^{13}	0	0	0
$\#S_{b,2}$	0	0	0	0	$\mathbf{2^{13}}$	2^{13}	2^{13}	2^{13}	0	0	0	0	2^{13}	2^{13}	2^{13}	2^{13}
$\#S_{b,3}$	0	0	0	2^{14}	$\mathbf{2^{14}}$	0	0	0	2^{13}	0	0	2^{13}	2^{13}	0	0	2^{13}

c	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$\#S_{c,0}$	0	0	2^{13}	0	$\mathbf{2^{13}}$	0	2^{13}	2^{13}	2^{13}	$\mathbf{2^{14}}$	0	0	0	0	0	2^{13}
$\#S_{c,1}$	0	2^{13}	0	2^{14}	$\mathbf{2^{14}}$	0	2^{13}	0	0	$\mathbf{2^{13}}$	0	0	0	0	2^{13}	0
$\#S_{c,2}$	0	2^{13}	0	0	$\mathbf{2^{13}}$	0	0	0	0	$\mathbf{2^{13}}$	2^{13}	2^{13}	2^{14}	2^{13}	0	0
$\#S_{c,3}$	0	0	2^{14}	0	$\mathbf{2^{13}}$	2^{13}	0	0	0	$\mathbf{2^{13}}$	2^{13}	0	0	2^{13}	0	2^{13}

d	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$\#S_{d,0}$	0	2^{13}	0	0	2^{13}	2^{13}	0	2^{13}	$\mathbf{2^{13}}$	2^{14}	0	0	0	0	0	2^{13}
$\#S_{d,1}$	0	2^{13}	0	0	0	0	2^{13}	0	$\mathbf{2^{14}}$	0	2^{13}	0	2^{13}	0	2^{13}	2^{13}
$\#S_{d,2}$	0	0	0	0	2^{14}	0	0	2^{14}	$\mathbf{2^{14}}$	0	0	2^{14}	0	0	0	0
$\#S_{d,3}$	0	2^{13}	2^{13}	0	0	0	2^{14}	0	$\mathbf{2^{14}}$	0	0	0	2^{13}	0	0	2^{13}

Those columns that have four non-zero entries contribute to the possible fault values and thus to the set of possible key candidates, as the correct secret key has to fullfil all equations simultaneously. Above, these columns are marked in **boldface**. From this we see that there are two fault tuples, namely $(9, 4, 4, 8)$ and $(9, 4, 9, 8)$. The corresponding key candidate sets both have 2^{24} elements, which results in an overall key space size of 2^{25} candidates.

The problematic equations are obviously equations $E_{c,i}$ for $i \in \{0, 1, 2, 3\}$. There are two possible fault values, namely 4 and 9. So far we have no way of deciding which set contains the key and thus have to search through both of them. Actually, in this example the correct key is contained in the candidates set corresponding to the fault tuple (9,4,4,8).

Key Set Filtering

In the following, we study the problem how to decide if a key candidate set contains the true key or not.

Let $x_i \in \mathbb{F}_{16}$ with $i \in \{0, 4, 8, 12\}$ be the elements of the first column of the state matrix at the beginning of round $r = 31$. The fault propagation in Figure 11 implies the following equations for the faulty elements x'_i:

$$x'_0 = x_0 + 4f' \qquad x'_8 = x_8 + Bf'$$
$$x'_4 = x_4 + 8f' \qquad x'_{12} = x_{12} + 2f' \ .$$

Next, let $y_i \in \mathbb{F}_{16}$ be the values that we get after adding the round constants to the elements x_i and plugging the result into the S-box. These values satisfy the following equations:

$$\begin{aligned} S(x_0 + \mathtt{0}) &= y_0 & S(x'_0 + \mathtt{0}) &= y_0 + a \\ S(x_4 + \mathtt{1}) &= y_4 & S(x'_4 + \mathtt{1}) &= y_4 + b \\ S(x_8 + \mathtt{2}) &= y_8 & S(x'_8 + \mathtt{2}) &= y_8 + c \\ S(x_{12} + \mathtt{3}) &= y_{12} & S(x'_{12} + \mathtt{3}) &= y_{12} + d \ . \end{aligned}$$

Now we apply the inverse S-box to these equations and take the XOR differences of the equations involving the same elements y_i, which results in the following system:

$$\begin{aligned} \mathtt{4}f' &= S^{-1}(y_0) + S^{-1}(y_0 + a) \\ \mathtt{8}f' &= S^{-1}(y_4) + S^{-1}(y_4 + b) \\ \mathtt{B}f' &= S^{-1}(y_8) + S^{-1}(y_8 + c) \\ \mathtt{2}f' &= S^{-1}(y_{12}) + S^{-1}(y_{12} + d) \ . \end{aligned}$$

Finally, we are ready to use a filter mechanism similar to the one in the preceding subsection. For a given fault tuple (a, b, c, d), we try all possible values of the elements y_i and check whether there is one for which the system has a solution for f'. Thus, we have to check four equations over $\mathbb{F}_{16}$ for consistency. This is easy enough and can also be done in parallel on multiple processors. If there is no solution for f', we discard the entire candidate set. We derived the equations above from a fault in entry s_0. However, we can model those equations also in a generic way, similar to the fault equations for key tuple filtering as given in Figure 13. Therefore, we introduce another set of fault coefficients e'_0, e'_1, e'_2 and e'_3, which depend on the fault location l and allow us to specify the generic representation of the fault equations for key set filtering, see Figure 14.

$$\begin{aligned} e'_0 \cdot f' &= S^{-1}(w) + S^{-1}(w + a) \\ e'_1 \cdot f' &= S^{-1}(x) + S^{-1}(x + b) \\ e'_2 \cdot f' &= S^{-1}(y) + S^{-1}(y + c) \\ e'_3 \cdot f' &= S^{-1}(z) + S^{-1}(z + d) \end{aligned}$$

Figure 14: The LED fault equations for key set filtering.

For the fault coefficients e'_0, e'_1, e'_2 and e'_3 we get once more four categories of possible assignments. These are shown in Figure 15.

Thus, if l is known, we just have to pick the corresponding assignments for the fault coefficients. However, if l is unknown we have to check each of the four possibilities, since

$$(e'_0, \ldots, e'_3) = \begin{cases} (4, 8, \mathtt{B}, 2), & \text{if } l \in \{0, 1, 2, 3\} \\ (8, 6, 5, 6), & \text{if } l \in \{4, 5, 6, 7\} \\ (\mathtt{B}, \mathtt{E}, \mathtt{A}, 9), & \text{if } l \in \{8, 9, 10, 11\} \\ (2, 2, \mathtt{F}, \mathtt{B}), & \text{if } l \in \{12, 13, 14, 15\} \end{cases}$$

Figure 15: Values of the fault coefficients for the LED key set filtering equations.

an incorrectly chosen assignment does not yield any solutions at all, only the correct one will generate solutions. Note that we are currently not using the absolute values for w, x, y, and z for the attack, but it might be possible to further speed-up filtering by integrating these values in the analysis.

Temporal and Spatial Aspects of the Filtering Mechanisms

The successful execution of the attack depends strongly on the fault injection in round 30:

1. Injecting the fault at an earlier round does not lead to useful fault equations, since they would depend on all key elements $k_0, \ldots, k_{15}$ and no meaningful key filtering would be possible.

2. Injecting the fault in a later round results in weaker fault equations which do not rule out enough key candidates to make exhaustive search feasible. As a consequence, more fault injections might be required to successfully reconstruct the secret key. Later we will describe in an other setting how to exploit multiple fault injections for secret key reconstruction.

3. Recall that the shape of the fault equations, see Figures 13 and 14, depends on the location l of the fault injection in round 30. Thus, if we allow fault injections at random (and unkown) entries of the state matrix in round 30, the overall time complexity rises by a factor of 4, since there are four fault location categories, as shown in Figure 12. If we also take key set filtering into account, the time complexity rises by a factor of 16, since there are an additional four categories for key set filtering, see Figure 15, which do not overlap with the ones from key tuple filtering.

Relationship to AES

Several properties of LED render it more resistant to the fault-based attack presented in this section, compared to AES as discussed in [191] and [235]. The derived LED fault equations are more complex than their counterparts for AES [191, 235]. This fact is due to

the diffusion property of the MixColumnsSerial function, which is a matrix multiplication that makes every block of the LED fault equations ($E_{x,j}$) (Section 2.4.2) depend on all 16 key indeterminates. In every block we have exactly one equation that depends on one of the key tuples (k_0, k_4, k_8, k_{12}), (k_1, k_5, k_9, k_{13}), $(k_2, k_6, k_{10}, k_{14})$, and $(k_3, k_7, k_{11}, k_{15})$. In contrast, AES skips the final MixColumns operation, and every block of its fault equations depends only on four key indeterminates.

Note that the fault attacks against AES, as described in [191, 235], would become much harder by adding the operation MixColumns to the last round of AES, since the time for evaluating the AES equations rises up to the factor 2^{32}. Furthermore, as in the case of LED, it is possible that several fault tuples have to be considered, further complicating the attack.

2.4.4 Experimental Results

In this section, we report on some results and timings of our attack. The timings were obtained on a 2.1 GHz AMD Opteron 6172 workstation having 48 GB RAM. The LED cipher was implemented in C, the attack code in Python. We performed our attack on $10,000$ examples using random keys, plaintext units, and faults. The faults were injected at the first element of the state matrix on the beginning of round $r = 30$. On average, it took about 45 seconds to finish a single run of the attack, including key tuple and key set filtering. The time for exhaustive search was not measured at this point. The execution time of the attack could be further reduced by using a better performing programming language like C/C++ and parallelization.

Table 6 shows the possible number of fault tuples (#ft) that appeared during our experiments and the relation between the number of occurrences and the cases where fault tuples could be discarded by key set filtering (Section 2.4.3). For instance, column 3 (#ft = 2) reports that there were 3926 cases in which two fault tuples were found and in 1640 cases one of them could be eliminated using key set filtering.

Table 6: Effects of key set filtering.

#ft	1	2	3	4	5	6	8	9	10	12	16	18	24	36
occurred	2952	3926	351	1887	1	307	394	15	1	101	39	10	14	2
discarded	0	1640	234	1410	1	268	359	14	1	101	38	10	14	2
ø discarded	-	0.4	0.9	1.4	2.0	2.5	3.6	3.7	5.0	6.1	8.4	8.4	12.6	24.0

It is clear that key set filtering is very efficient. Especially if many fault tuples had to be considered, some of them could be discarded in almost every case. But also in the more frequent case of a small number of fault tuples there was a significant gain. Figure 16 shows this using a graphical representation (note the logarithmic y scale). Altogether, in about 29.5% of the examples there was a unique fault tuple, in another 29.6% of the

examples there were multiple fault tuples, none of which could be discarded, and in about 40.9% of the examples some of the fault tuples could be eliminated using key set filtering.

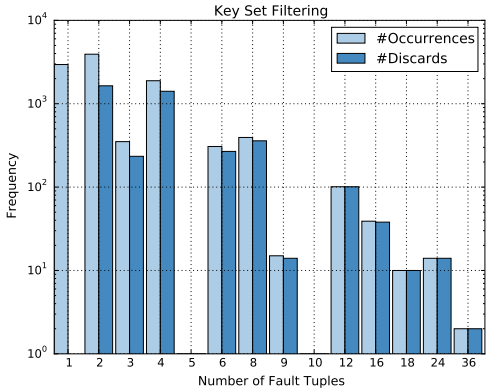


Figure 16: Efficiency of key set filtering.

2.4.5 Extensions of the Fault Attack

In this section, we discuss some improvements and extensions of the attack as introduced in Section 2.4.3.

Diagonal Fault Injections

The class of diagonal faults is a generalisation to faults injected only in a single state element. Diagonal faults were introduced in [216] to attack AES. As the name suggests, these faults are injected into the diagonal of the state matrix, so that the ShiftRows operation moves the faulty elements to the same column. In the subsequent MixColumns operation the faults are then still limited to the same column. Thus, diagonal faults show the same fault propagation behaviour as faults injected only into a single element of the state. Due to the similarity of LED to AES, it is not surprising that diagonal fault injections are also applicable to LED and thus give an attacker some additional degree of freedom without increasing the cost of the subsequent analysis. Figure 17 shows an example of four diagonal faults injected in round 30 of the LED-64 encryption that exhibit the same fault propagation behaviour in rounds 31 and 32. For a 4×4 matrix and a fixed diagonal there are

$$\binom{4}{1} + \binom{4}{2} + \binom{4}{3} + \binom{4}{4} = 15$$

different fault patterns that all share the same propagation behaviour. Furthermore, Figure 18 shows all of the four possible diagonal fault equivalence classes.

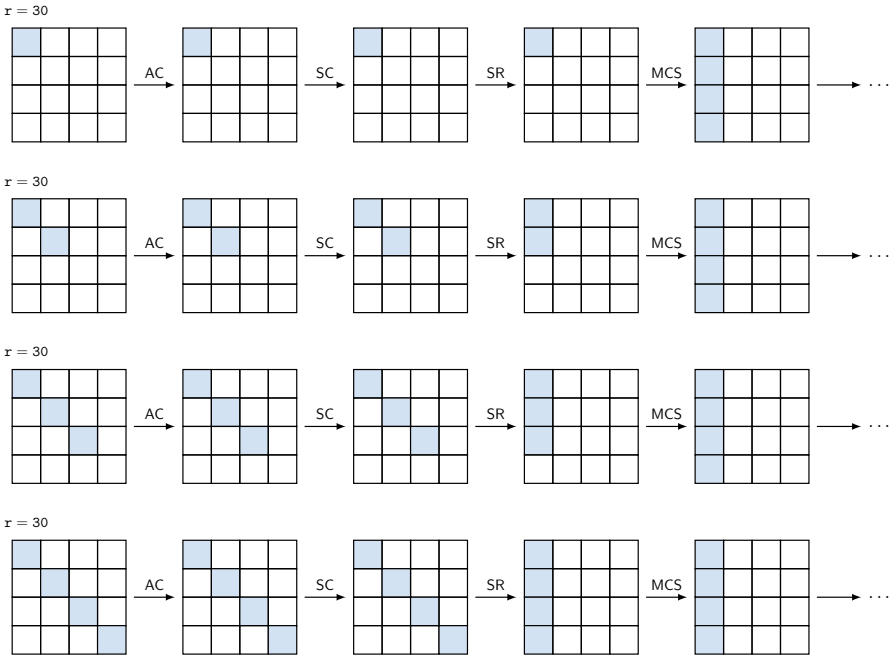


Figure 17: Diagonal faults in LED with the same fault propagation behaviour.

Multiple Fault Injections

It is possible to further reduce the key space by running the attack a second time with the same key but a different plaintext or different fault value. Since the correct key has to be in one of the candidate sets of both runs, a pairwise intersection of the key candidate sets from the first and the second attack eliminates many "wrong" candidate sets, on the one hand, and, on the other, greatly reduces the number of candidates in the correct one. The following example illustrates this technique.

We repeat the attack from Section 2.4.3 with the same key K and plaintext M but a different fault having the value $e = \texttt{A}$. It is again injected in the first entry of the state at the beginning of round $r = 30$. This results in the following setup:

$$\begin{aligned} K &= \texttt{01234567 89ABCDEF} \\ M &= \texttt{01234567 89ABCDEF} \\ C &= \texttt{A003551E 3893FC58} \\ C'' &= \texttt{4FF1AA93 72F38074}\ . \end{aligned}$$

The key filtering stage returns one fault tuple $(\texttt{3}, \texttt{1}, \texttt{6}, \texttt{1})$ and the corresponding key candidate set has a size of 2^{21}. Now we form the pairwise intersections of the key candidate

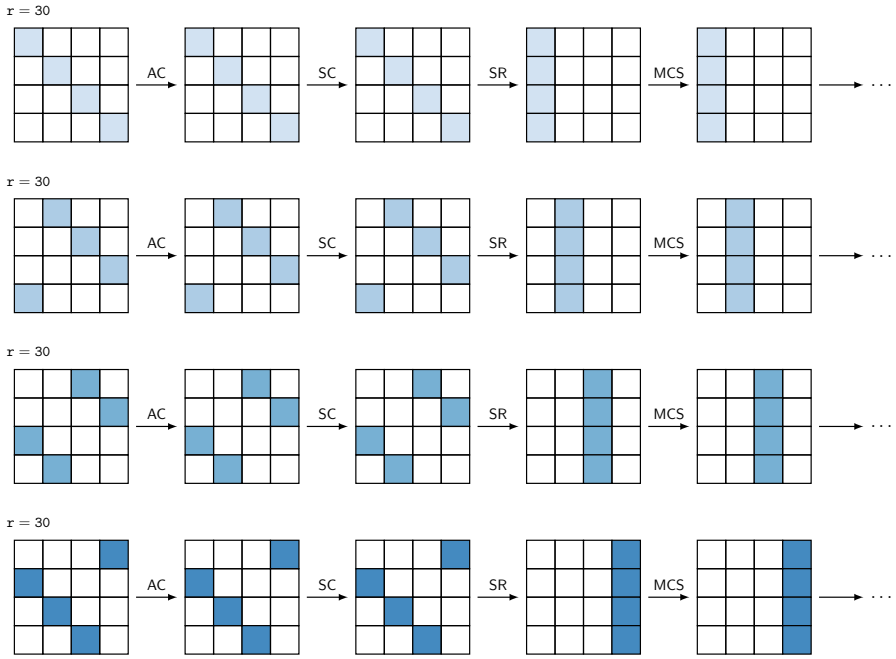


Figure 18: Diagonal fault equivalence classes for LED.

sets of the first and second run. The only non-empty one contains just 1 key, which is also the correct secret key. However, it might be not always the case that just a single key remains after the computation of the intersections. Note that the attack would also work if we vary the plaintext instead of the fault value. Obviously, only the key needs to stay fixed.

A repetition of an attack may or may not be feasible in practice. Experiments demonstrate that our technique works using a single attack; several attacks just further reduce the set of key candidates on which to run an exhaustive search.

Extension of the Attack to LED-128

Recall that LED-128 uses a 128-bit key K which is split into two 64-bit subkeys K_1 and K_2 used alternatingly as round keys, see Figure 8. Since K_1 and K_2 are independent from each other, a straightforward application of the procedure would result in fault equations with too many indeterminates to allow sufficient key filtering. Unlike AES (where reconstruction of the last subkey allows the derivation of all other subkeys from the (bijective) key schedule [191]), LED-128 inherently resists the fault attack under the assumptions of this chapter.

Still, LED-128 is vulnerable to a fault attack if we assume that the attacker has the

capability assumed in previous literature ([168], p. 298). If the key is stored in a secure memory (EEPROM) and transferred to the device's main memory when needed, the attacker may reset selected bytes of the key, i.e., assign them the value 0, during the transfer from the EEPROM to the memory. If we can temporary set the round key K_2 to 0 (or any other known value) and leave K_1 unchanged, then a simple modification of our attack can derive K_1. Using the knowledge of K_1, we mount a second fault attack without manipulating K_2. This second attack is another modification of our attack and is used to determine K_2.

The attacker described in the attack on LED-128 has to be obviously very strong and would not adhere to the fault models as introduced in Section 2.4.1, which is a major disadvantage of this approach. In the next section, we extend the ideas from above and derive a framework which allows to attack SPN block ciphers with independent subkeys. This framework enables us to execute attacks on LED-128 and PRINCE under the more realistic fault models RKF and RUF.

2.5 Multi-Stage Fault Attacks on LED-128 and PRINCE

2.5.1 The Multi-Stage Fault Attack Framework

Let $\mathcal{E}$ be a block cipher with block size 2^n and $n \in \mathbb{N}$. We assume that the secret key k is written as a concatenation of independent subkeys $k = k_0 \parallel k_1 \parallel \cdots \parallel k_{s-1}$ such that each subkey is of size 2^n. Further, we suppose that the encryption algorithm works on 2^m-bit sized parts (for $m = 2$ those are called *nibbles*) of the 2^n-bit state. For the 64-bit ciphers LED-128 and PRINCE two subkeys are used and we have $n = 6$, $s = 2$, and $m = 2$. The proposed Multi-Stage Fault Attack is a known-plaintext attack, proceeds in s *stages* (one stage per subkey) and its realisation in pseudo-code is illustrated in Algorithm 3.

The attack starts by encrypting a known plaintext p in the absence of faults and by recording the obtained ciphertext c through querying an encryption oracle, which is denoted by $c = \mathcal{O}_k^{\mathcal{E}}(p)$, see Step 2. The oracle $\mathcal{O}_k^{\mathcal{E}}$ symbolizes the attacked black box device which, when supplied with a message p, returns its encrypted version c using the secret key k. It is assumed that k is stored on the device and the attacker has no access to it.

Each stage $i \in \{0, 1, \ldots, s-1\}$ consists of one or several fault injections. We enumerate the fault injections by $j \in \mathbb{N}$. The particular parameters of a fault injection (location and time) depend on the cipher under analysis and on the capabilities of the attacker. For example, ciphers based on substitution-permutation networks typically require fault injection two rounds before termination for successful differential fault analysis. Let f_{ij} denote the fault induced during the jth fault injection of stage i and let c_{ij} be the ciphertext obtained by a fault-affected encryption of p using k. This operation is denoted through the oracle query $c_{ij} = \mathcal{O}_{k,f_{ij}}^{\mathcal{E}}(p)$, see Step 8. Note that c_{ij} is observable by the attacker, who is assumed to have physical access to the hardware into which he injects

faults. However, he may or may not know which fault f_{ij} was injected, as many physical fault-injection techniques do not allow perfect control of the bits that flip as a result of the disturbance. For more details on the fault models, refer to Section 2.4.1. Moreover, Figure 19 shows an overview of multi-stage fault attacks on SPN block ciphers.

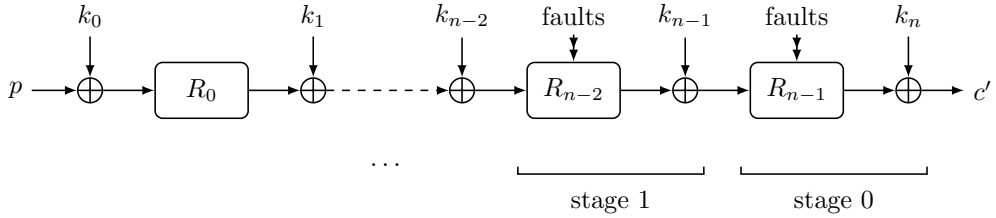


Figure 19: Overview on multi-stage fault attacks on SPN block ciphers.

We denote by $\mathsf{analyse}(c, c_{ij})$, as used in Steps 10 and 15, a cipher-dependent procedure that performs differential fault analysis and yields a set K_{ij} of candidates for the ith part k_i of the secret key. We will introduce two instances of the procedure $\mathsf{analyse}(c, c_{ij})$ for the ciphers LED-128 in Section 2.5.2 and PRINCE in Section 2.5.4. Performing multiple fault injections during the same stage results in multiple invocations of $\mathsf{analyse}(c, c_{ij})$ for different c_{ij} and therefore results in different sets of subkey candidates K_{ij}. Since the correct subkey k_i must be contained in all K_{ij}, it must also be contained in their intersection $K_i = \bigcap K_{ij}$ which is frequently much smaller than the individual sets K_{ij}. Consequently, multiple fault injections reduce the size of the subkey candidate set. Note that no reduction occurs if the same fault is injected multiple times, resulting in identical c_{ij}; in that case, the fault injection must be repeated.

After all stages have been performed, the final candidate set can be obtained by computing the Cartesian product $K = K_0 \times \cdots \times K_{s-1}$, where K_i are subkey candidate sets calculated during the individual stages. As it will become apparent, it is possible to further reduce this set and therefore the complexity of the subsequent brute-force search.

The Multi-Stage Fault Attack algorithm incorporates a mechanism to balance between the available computational power and the number of fault injections necessary in order to successfully execute the attack. As was observed above, more fault injections in stage i will reduce the set of subkey candidates K_i. Let T be an estimate of time complexity of one invocation of procedure $\mathsf{analyse}(c, c_{ij})$ (either actual run-time in milliseconds or number of operations). We define a sequence of *threshold values* $\tau_0, \ldots, \tau_{s-1}$ which have the same unit as T. The value of τ_i roughly represents the amount of computational power allocated to stage i; it will be used to set an upper bound for the number of invocations of procedure $\mathsf{analyse}(c, c_{ij})$ in stage i. In stage 0 at the beginning of the algorithm, fault injections are continued until subkey candidate set K_0 becomes so small that $T \cdot \#K_0 < \tau_0$ holds. In stage i, subkey candidate sets $K_0, \ldots, K_{i-1}$ have been calculated already. Each subkey combination $(k_0, \ldots, k_{i-1}) \in K_0 \times \cdots \times K_{i-1}$ is used to partially decrypt c and c_{ij}

(see Steps 13 and 14) to obtain intermediate states v and v_x. Then analyse is applied on v and v_x to construct candidates for k_i. The worst-case number of procedure invocations is $\#(K_0 \times \cdots \times K_{i-1})$, and the number of fault injections is set such that K_i is sufficiently small to fulfil $T \cdot \#(K_0 \times \cdots \times K_i) < \tau_i$. In Step 24, we can leverage the knowledge which x led to which key candidate set K_{ij} by computing $\{x\} \times K_{ij}$ instead of the complete Cartesian product, which obviously keeps the size of the final key candidate set K smaller.

2.5.2 Applications to LED-128

The attack on LED-128 is based on the differential fault analysis of LED-64, as introduced in Section 2.4. Recall that it employs one fault injection and reduces the number of key candidates to 2^{19} – 2^{26}, a number that can be handled by exhaustive search. Applying this attack to LED-128 would shrink the key space for the last applied subkey k_0, but considering all combinations of up to 2^{26} candidates for k_0 and 2^{64} possibilities for the second subkey k_1 is clearly infeasible. The Multi-Stage Fault Attack algorithm solves this problem in two stages $i \in \{0, 1\}$, one for the reconstruction of each 64-bit subkey $k_i = k_{i,0} \parallel \cdots \parallel k_{i,15}$ with nibbles $k_{i,0}, \ldots, k_{i,15}$. The filtering mechanism is similar to the one introduced in Section 2.4.3 and reuses the fault equations from Figure 13. However, the inputs to the analyse function varies depending on the stage in the Multi-Stage Fault Attack algorithm.

In more detail, to run the Multi-Stage Fault Attack on LED-128, we first produce faulty ciphertexts c'_0 in stage 0, by injecting faults in round $r_0 = 46$. Then we apply analyse to the pair of ciphertexts c and c'_0 which generates a set of key candidates K_0. Recall, as described in Section 2.5.1, that multiple fault injections and thus multiple calls to analyse might be necessary until the set K_0 is smaller than the initially defined threshold τ_0. In stage 1 we inject faults in round $r_1 = 42$ to get faulty ciphertexts c'_1. Then, for each $x \in K_0$, we partially decrypt c and c'_1 by 4 rounds and apply analyse to each of those pairs which results in a key candidate set $K_1(x)$. The union of the sets $K_1(x)$ forms the candidate set K_1 of the subkey k_1. As in stage 0, multiple fault injections might be necessary to shrink K_1 until its size is smaller than τ_1. The final step of the attack determines the correct key by a brute-force search on $K_0 \times K_1$.

Complexity Analysis of the Attack

Executing one run of the analyse method under the RKF model requires 2^{20} evaluations of single fault equations, as each one of the 16 fault equations $E_{x,i}$ (see Figure 13), with $x \in \{a, b, c, d\}$ and $i \in \{0, 1, 2, 3\}$, depends on four key nibbles. Compared to that, the evaluation of the fault equations for key set filtering (see Figure 14) has negligible time complexity and hence is not considered further. The complexity for the RUF model is even higher with 2^{24} evaluations. Under that model, the location of the fault injection is unknown and an attacker has to try all 16 possible combinations for the possible

Algorithm 3: multi_stage_fault_attack$[s](p, \tau, T)$

Inputs:

p, a plaintext

$\tau = (\tau_0, \ldots, \tau_{s-1})$, a sequence of thresholds

T, time complexity limit of analyse

Output:

k, the secret key

Algorithm:

1. $i \leftarrow 0,\ j \leftarrow 0,\ K \leftarrow \emptyset$
2. $c \leftarrow \mathcal{O}_k^{\mathcal{E}}(p)$
3. **while** $i < s$ **do**
4. $\quad K_i \leftarrow \emptyset$
5. $\quad$ **while** $\tau_i \leq T \cdot \#(K \times K_i)$ **or** $K = \emptyset$ **do**
6. $\quad\quad K_{ij} \leftarrow \emptyset$
7. $\quad\quad f_{ij} \xleftarrow{\$} \mathbb{F}_2^n$
8. $\quad\quad c_{ij} \leftarrow \mathcal{O}_{k,f_{ij}}^{\mathcal{E}}(p)$
9. $\quad\quad$ **if** $K = \emptyset$ **then**
10. $\quad\quad\quad K_{ij} \leftarrow K_{ij} \cup \mathsf{analyse}(c, c_{ij})$
11. $\quad\quad$ **else**
12. $\quad\quad\quad$ **for** $x \in K$ **do**
13. $\quad\quad\quad\quad v \leftarrow \mathsf{partial_decrypt}_x(c)$
14. $\quad\quad\quad\quad v_x \leftarrow \mathsf{partial_decrypt}_x(c_{ij})$
15. $\quad\quad\quad\quad K_{ij} \leftarrow K_{ij} \cup \mathsf{analyse}(v, v_x)$
16. $\quad\quad\quad$ **end**
17. $\quad\quad$ **end**
18. $\quad\quad$ **if** $j > 0$ **then**
19. $\quad\quad\quad K_{ij} \leftarrow K_i \cap K_{ij}$
20. $\quad\quad$ **end**
21. $\quad\quad K_i \leftarrow K_{ij}$
22. $\quad\quad j \leftarrow j + 1$
23. $\quad$ **end**
24. $\quad K \leftarrow K \times K_i$
25. $\quad i \leftarrow i + 1,\ j \leftarrow 0$
26. **end**
27. **for** $k \in K$ **do**
28. $\quad$ **if** $c = \mathsf{encrypt}_k(p)$ **then**
29. $\quad\quad$ **return** k
30. $\quad$ **end**
31. **end**
32. **return** ε

assignments of the fault coefficients $e_0, \ldots, e_{15}$ (Figure 12) and $e'_0, \ldots, e'_3$ (Figure 15). This results in a complexity of about $T \cdot 2^{20}$ (for RKF) respectively $T \cdot 2^{24}$ (for RUF) for stage 1 where T is the number of key candidates for k_0.

2.5.3 Experimental Results

The results of the Multi-Stage Fault Attack on LED-128 were obtained from 10,000 runs of the attack using the RUF model. Figure 20 shows the frequency for the numbers of remaining key candidates after 1 and 2 fault injections for stage $i \in \{0, 1\}$. The number of times a particular count of key candidates appeared during our experiments is displayed on the Y-axis. Table 7 shows the numbers of key candidates if 1 and 2 fault injections are performed in stage 0 and 1 of the attack. If the attacker injects just a single fault in stage 0 he has to execute, on average, $2^{24} \cdot 2^{23.64} = 2^{47.64}$ evaluations of single fault equations in stage 1. Even in the rare best case where stage 0 yields 2^{17} key candidates, the complete attack would require 2^{41} evaluations, which is a rather high value if only a medium amount of computational power is available. However, as soon as we inject a second fault in stage 0, the number of key candidates for k_0 drops very rapidly to an average value of $2^{3.26}$ which makes the analysis feasible even on slow hardware. Thus, 3 fault injections are required to break LED-128 on average: 2 in stage 0 and 1 in stage 1.

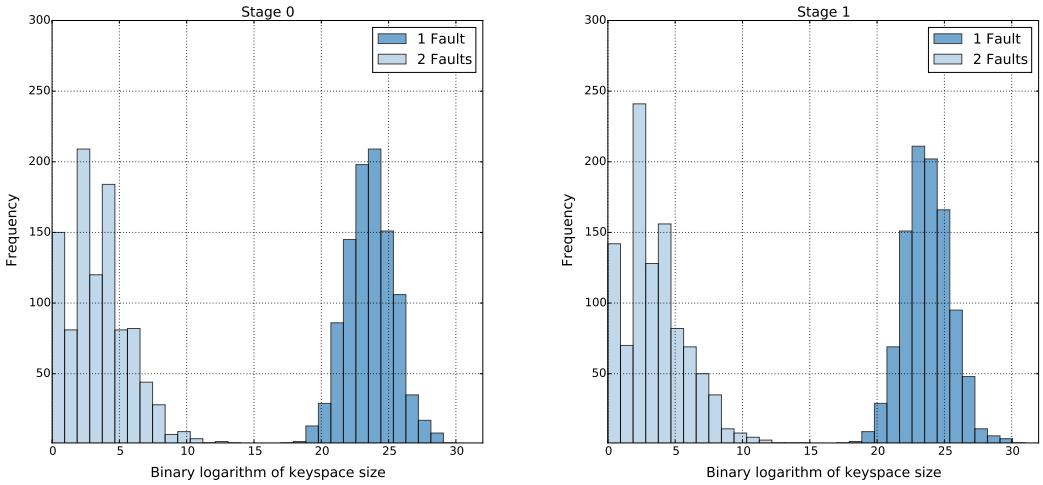


Figure 20: Frequency of LED-128 key set sizes in stages 0 and 1.

Table 7: Number of remaining candidates for k_i in the LED-128 attack with $i \in \{0, 1\}$.

	#keys after stage 0		#keys after stage 1	
#faults	1	2	1	2
min	$2^{17.00}$	1	$2^{17.00}$	1
max	$2^{30.00}$	$2^{14.00}$	$2^{31.00}$	$2^{15.00}$
avg	$2^{23.64}$	$2^{3.26}$	$2^{23.71}$	$2^{3.32}$
median	$2^{24.50}$	$2^{8.00}$	$2^{25.00}$	$2^{8.50}$

2.5.4 Applications to PRINCE

The fault attack[2] on PRINCE requires two stages, as k_0 can be easily derived as soon as k_2 is known. First, we discuss the functionality of the analyse method. To produce faulty ciphertexts for stage 0, we inject faults between the application of the S-Layer in round R_8^{-1} and the multiplication with the matrix M' in round R_9^{-1}. For stage 1, we inject faults exactly one round earlier. Figure 21 shows the fault propagation of a fault in PRINCE over two R^{-1} rounds.

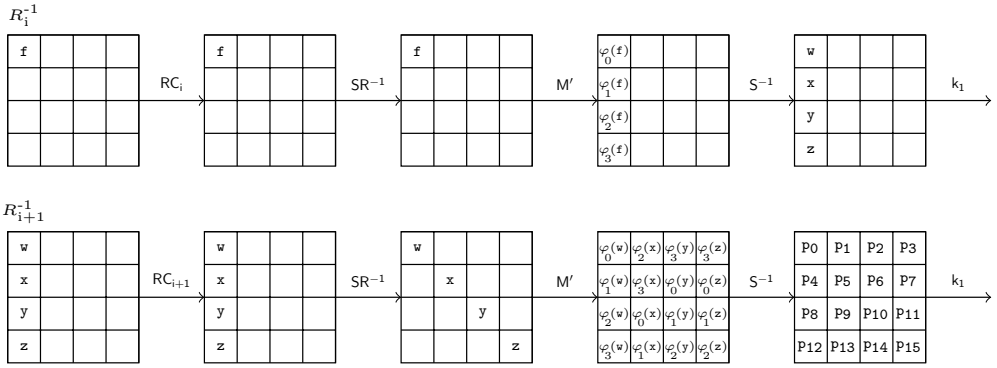


Figure 21: Fault propagation in PRINCE over two R^{-1} rounds.

In order to construct the fault equations used for key candidate filtering, we start, as in the case of LED, with the correct and faulty ciphertexts $c = c_0 \parallel \cdots \parallel c_{15}$ and $c' = c'_0 \parallel \cdots \parallel c'_{15}$ (or with the respective intermediate states in stage 1) and work backward through the encryption, inverting each of the steps, up to the point before the application of the last S-box. This corresponds to the last "matrix" in Figure 21. Let us start by fixing notation.

[2]Note that a similar fault-based attack on PRINCE was developed independently in [228].

Let $i \in \{0, \ldots, 15\}$ and let v_i and v_i' be variables representing the ith nibble of the correct and the faulty ciphertext (or the ith correct and faulty state nibble in case of stage 1 of the attack), respectively. Moreover, the variables p_i represent key nibbles and q_i round constants. In stage 0, we substitute the nibbles of RC_{11} for q_i and in stage 1 the nibbles of RC_{10} for q_i.

Let us point out a particular feature of the attack on PRINCE: an adversary cannot reconstruct one of the secret keys directly during stage 0 of the attack. The keys k_1 and k_2 are applied immediately in succession during one run of the encryption (see Figure 10) and thus we can only reconstruct the XOR value $k_1 \oplus k_2$. However, this fact presents no drawback for the feasibility of the attack. In stage 1, we directly reconstruct candidates for k_1 which then obviously allows to derive k_2 from $k_1 \oplus k_2$.

Let $a = b_0 \parallel b_1 \parallel b_2 \parallel b_3$ be a 4-bit value and $j \in \{0, \ldots, 3\}$. Then we define $\varphi_j(a)$ as the value equal to a where the jth bit b_j is set to 0. So, for example, for $j = 2$ we get $\varphi_2(a) = b_0 \parallel b_1 \parallel 0 \parallel b_3$. Let w, x, y, and z be variables and let $j_i \in \{0, \ldots, 3\}$ then we can describe the fault equations E_i of PRINCE as outlined in Figure 22.

$$E_i: \; S(v_i \oplus p_i \oplus q_i) \oplus S(v_i' \oplus p_i \oplus q_i) = \begin{cases} \varphi_{j_i}(w), & i \in \{0, \ldots, 3\} \\ \varphi_{j_i}(x), & i \in \{4, \ldots, 7\} \\ \varphi_{j_i}(y), & i \in \{8, \ldots, 11\} \\ \varphi_{j_i}(z), & i \in \{12, \ldots, 15\} \end{cases}$$

Figure 22: Fault equations for PRINCE.

The values of the indices j_i from the variables on the right-hand sides of the equations E_i are derived from the multiplication of the state with the matrix M' (see Figure 21) and depend on the location l of the injected fault. The possible assignments are listed in Figure 23.

$$(j_0, \ldots, j_{15}) = \begin{cases} (0,1,2,3,2,3,0,1,3,0,1,2,3,0,1,2), & \text{if } l \in \{0,7,10,13\} \\ (3,0,1,2,1,2,3,0,2,3,0,1,2,3,0,1), & \text{if } l \in \{1,4,11,14\} \\ (2,3,0,1,0,1,2,3,1,2,3,0,1,2,3,0), & \text{if } l \in \{2,5,8,15\} \\ (1,2,3,0,3,0,1,2,0,1,2,3,0,1,2,3), & \text{if } l \in \{3,6,8,12\} \end{cases}$$

Figure 23: Fault location dependent values of the indices j_i for the equations E_i.

A 4-bit value t is called *valid with respect to pattern* φ_{j_i} if the binary representation of

t has a 0 at the bit position j_i. In the following, we use bit pattern matching to construct inductively a set S_i which will ultimately contain candidates for the nibble p_i of the subkey. Key candidate filtering is done in three steps: *Evaluation*, *Inner Filtering*, and *Outer Filtering* which are described next. Note that the first two steps could be executed together, but for better comprehensibility we describe them separately.

Evaluation

Each equation E_i is evaluated for all possible 4-bit values u of nibble candidates associated to the variable p_i. When the result of an evaluation $t = E_i(u)$ has been computed, the tuple (t, u) is appended to the set S_i.

Inner Filtering

In this step, we check for all tuples $(t, u) \in S_i$ if the entry t is valid with respect to the bit pattern φ_{j_i}. Those tuples that do not have a valid entry t are discarded, all others are kept.

For example, we take fault equation E_0 and assume that a fault was injected in nibble $l = 0$. From the definition of j_i above we see that the 0th entry of j_0 is 0. Moreover, we assume that the tuple $(t, u) = (7, 3)$ is an element of S_0 and observe immediately that 7 matches the bit pattern $\varphi_{j_0} = 0 \parallel s_1 \parallel s_2 \parallel s_3$. Thus, $(7, 3)$ is a valid tuple and 3 a potential candidate for the nibble associated to p_0.

Outer Filtering

The idea in the final filtering step is to exploit the fact that the elements of the sets $S_{4m}, \ldots, S_{4m+3}$ are related to each other for a fixed $m \in \{0, \ldots, 3\}$. This is due to the fact that the right-hand sides of the equations $E_{4m}, \ldots, E_{4m+3}$ are derived from a common pre-image. This can be utilized to build conditions for filtering candidates of the nibbles associated to $p_{4m}, \ldots, p_{4m+3}$. First, we fix $m \in \{0, \ldots, 3\}$ and order the tuples $(t_{4m+n}, u_{4m+n}) \in S_{4m+n}$ lexicographically for all $n \in \{0, \ldots, 3\}$. Next, we compute the sets P_{4m+n} containing the pre-images of all the values t_{4m+n}. This is done as follows. After the Inner Filtering, all values t_{4m+n} match the bit pattern derived from φ_{4m+n}. However, we do not know if the j_{4m+n}th bit of t_{4m+n} had value 0 or 1 before it was fixed to 0. Hence, we obviously have two possible values for the pre-images of t_{4m+n}. One is t_{4m+n} itself and the other has a 1 at bit position j_{4m+n}. Then we intersect the pre-image sets $P_{4m}, \ldots, P_{4m+3}$ with each other and obtain a set G_m of pre-image candidates. After that, we check for each $g_m \in G_m$ if, for every $n \in \{0, \ldots, 3\}$, there is at least one tuple in S_{4m+n} which has the value $\varphi_{i_{4m+n}}(g_m)$ in its first component. If so, g_m is a valid pre-image. When all pre-images have been processed, all tuples are deleted from the sets $S_{4m}, \ldots, S_{4m+3}$, except those where the first entry has a valid pre-image g_m. This finishes the filtering stage.

Finally, after projecting the sets S_i to their second components u_i, the Cartesian product over those projections is computed to get the key candidates for $k_1 \oplus k_2$. If the number of candidates for $k_1 \oplus k_2$ is small enough, i.e., if it falls below the threshold τ_0, stage 0 of the attack ends. Otherwise, the procedure above is repeated as described in the Multi-Stage Fault Attack algorithm in Section 2.5.1.

In the second stage of the attack, candidates for k_1 are computed using the previously described configurations for the fault injections. This is repeated until the number of candidates for k_1 falls below the specified threshold value τ_1. As soon as this is the case, the candidates for k_1 and $k_1 \oplus k_2$ are used to derive candidates for the subkeys k_2 and k_0. Finally, a brute-force search on the Cartesian product $K_0 \times K_1$ is performed to find the actual key $k_0 \parallel k_1$.

Complexity Analysis of the Attack

The complexity of the analyse method in the case of PRINCE is very low. The computationally most expensive part is the evaluation of the fault equations. For each of the 16 equations we have to compute 16 evaluations, since there are 16 possible values for the key nibbles. Altogether, this results in $2^8 = 256$ evaluations. As for the attacks on LED, the number of evaluations is multiplied by a constant factor when using the RUF fault model. In the case of PRINCE, this factor has the value 4, as there are four different bit patterns for key candidate filtering and the attacker does not know which pattern is the correct one, due to the unknown location of the fault injection. Therefore, all four patterns j_i (see Figure 23) have to be tried, which gives $2^{10} = 1024$ evaluations for one run of analyse. In stage 1, the analyse method may be executed up to T times in the worst case, where T is the number of $k_1 \oplus k_2$ candidates. This results in a complexity of about $2^8 \cdot T$ or $2^{10} \cdot T$ evaluations, depending on the fault model which has been used.

2.5.5 Experimental Results

In this section, we describe the results of the Multi-Stage Fault Attack on PRINCE. All results were obtained from 10,000 runs of the attack. For the computation of the results, the weaker RUF fault model was used. We observed that the differences in the sizes of the candidate sets between the fault models RKF and RUF were effectively non-existent. This can be explained by the observation that in almost all cases the candidate sets are empty when a wrong bit pattern $(j_i)_{i=0,\ldots,15}$ is used for filtering. Figure 24 gives an overview (with stacked bars) on the two stages of the attack for multiple fault injections and Table 8 summarizes the results of the attack on PRINCE.

For stage 0, we get on average $2^{30.89}$ candidates when injecting a single fault. This results in an overall complexity of $2^{40.89}$ evaluations of single fault equations for stage 1 under the RUF model. This is hardly feasible on common hardware. But with a second fault injection in stage 0, the complexity drops to $2^{11.44}$ evaluations, which is practically

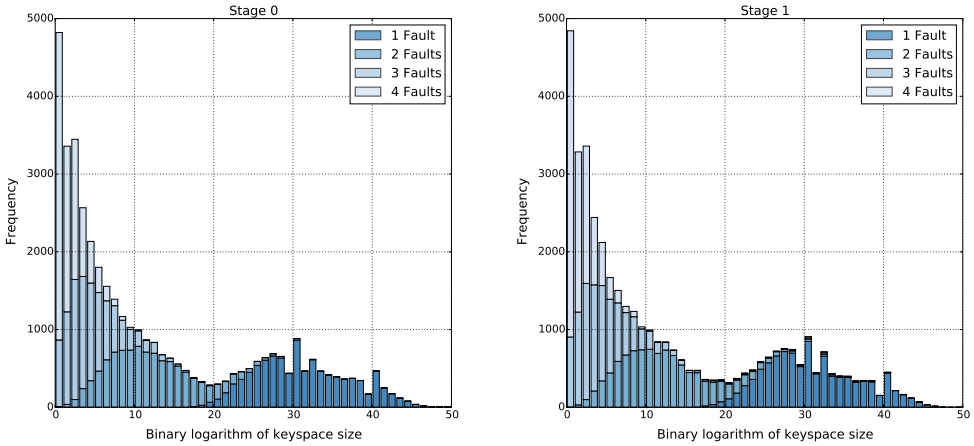


Figure 24: Frequency of PRINCE key set sizes in stages 0 and 1.

Table 8: Statistics for the number of candidates for $k_1 \oplus k_2$ and k_1.

	#keys after stage 0				#keys after stage 1			
#faults	1	2	3	4	1	2	3	4
min	$2^{17.00}$	1	1	1	$2^{16.00}$	1	1	1
max	$2^{50.00}$	$2^{38.00}$	$2^{24.00}$	$2^{12.00}$	$2^{49.00}$	$2^{44.00}$	$2^{40.00}$	$2^{43.00}$
avg	$2^{30.89}$	$2^{11.44}$	$2^{4.12}$	$2^{1.47}$	$2^{30.41}$	$2^{11.64}$	$2^{4.44}$	$2^{1.82}$
median	$2^{34.50}$	$2^{19.50}$	$2^{12.50}$	$2^{7.00}$	$2^{33.50}$	$2^{21.50}$	$2^{21.00}$	$2^{21.00}$

doable. Thus, on average, we need about two fault injections in stage 0 to be able to finish stage 1. Furthermore, the numbers for stage 0 do not differ significantly from those of stage 1, except for the median, which surprisingly stays rather constant after the first fault injection. But as we only have to search through the generated key candidate sets, the average of $2^{30.41}$ is feasible for brute-force. Nevertheless, note that the maximal sizes of the key candidate sets are still quite high. Thus there might be cases where more than one fault injection is required for stage 1. In summary, our experiments show that on average 3 to 4 faults are required to reconstruct the complete 128-bit key of PRINCE with a Multi-Stage Fault Attack on common hardware.

2.5.6 Extensions of the Fault Attacks

The attacks presented on LED-128 and PRINCE can be extended to the setting of diagonal faults analogically to LED-64 as discussed in Section 2.4.5. Recall, that diagonal faults give

an attacker some additional freedom during the fault injection phase, without affecting the subsequent analysis. Since the structural differences between LED-64 and LED-128 are only minor ones, the same fault propagation patterns as presented in Figures 17 and 18 apply to LED-128. For PRINCE, the four diagonal fault equivalence classes are depicted in Figure 25.

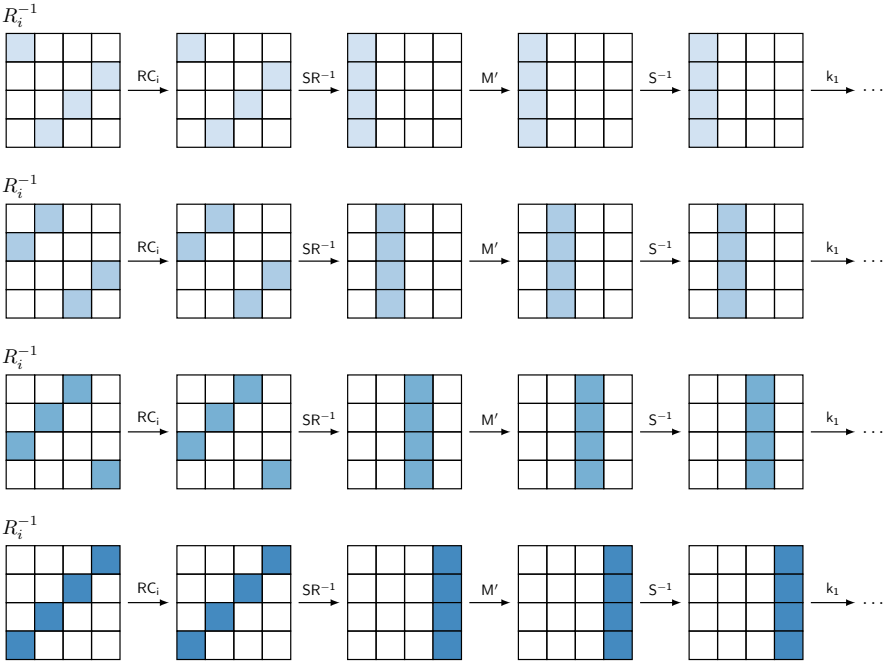


Figure 25: Diagonal fault equivalence classes for PRINCE.

2.6 Algebraic Fault Attacks on LED-64

In this section, we investigate an alternative approach to fault analysis, where we use algebraic techniques instead of differential fault analysis to search for the secret key. We also discuss the pros and cons of an algebraic solving strategy. We use the fault attack on LED-64 as a basis for our investigations. Before going into the details of the attack we first show how to model LED algebraically.

2.6.1 Algebraic Representation of LED

To get an algebraic representation of the LED cipher, we show for each step of the encryption algorithm how to model it via multivariate polynomials over $\mathbb{F}_2$. For the operations we refer to Section 2.2, where we described the specification of LED.

Recall that the 64-bit state s of the cipher is divided into 16 nibbles (4-bit strings) $s = s_0 \parallel s_1 \parallel \cdots \parallel s_{15}$ and these are arranged in a matrix of size 4×4 of the shape

$$s = \begin{pmatrix} s_0 & s_1 & s_2 & s_3 \\ s_4 & s_5 & s_6 & s_7 \\ s_8 & s_9 & s_{10} & s_{11} \\ s_{12} & s_{13} & s_{14} & s_{15} \end{pmatrix} .$$

Moreover, each 4-bit sized entry $s_i = a_{4i} \parallel a_{4i+1} \parallel a_{4i+2} \parallel a_{4i+3}$, with $0 \leq i \leq 15$, is identified with an element of the finite field $\mathbb{F}_{16} \cong \mathbb{F}_2[x]/\langle x^4 + x + 1\rangle$, as shown in Section 2.2.1. The residue classes of $\{1, x, x^2, x^3\}$ form an $\mathbb{F}_2$-vector space basis of this field. Then s_i corresponds to the field element $a_{4i}x^3 + a_{4i+1}x^2 + a_{4i+2}x + a_{4i+3}$ where $a_j \in \mathbb{F}_2$. In particular, if we combine the input bits to field elements $m_i = p_{4i}x^3 + p_{4i+1}x^2 + p_{4i+2}x + p_{4i+3}$, the input state of the encryption map is represented by the matrix

$$M = \begin{pmatrix} m_0 & m_1 & m_2 & m_3 \\ m_4 & m_5 & m_6 & m_7 \\ m_8 & m_9 & m_{10} & m_{11} \\ m_{12} & m_{13} & m_{14} & m_{15} \end{pmatrix}$$

of size 4×4 over the field $\mathbb{F}_{16}$. Similarly, we can represent the key by a matrix K (or two matrices K_1 and K_2 in case of LED-128) of size 4×4 over $\mathbb{F}_{16}$.

To construct the polynomial representation of LED, we use indeterminates $p_0, \ldots, p_{63}$ representing the bits of the plaintext, indeterminates $k_0, \ldots, k_{63}$ (LED-64) or $k_0, \ldots, k_{127}$ (LED-128) representing the key bits, and indeterminates $c_0, \ldots, c_{63}$ representing the bits of a ciphertext unit. The indeterminates $x_i^{(r)}$, $y_i^{(r)}$, and $z_i^{(r)}$ represent the state of the cipher after the operations AC, SR, and MCS during encryption round r, with more details following below, where we construct the polynomial representation for each operation, as introduced in Section 2.2.2. In case of LED-64 it is contained in the polynomial ring

$$\mathbb{F}_2[p_i, k_i, x_i^{(r)}, y_i^{(r)} z_i^{(r)}, c_i \mid i = 0, \ldots, 63; r = 1, \ldots, 32]$$

which has no less than 6336 indeterminates. For LED-128 we require an additional 64 indeterminates $k_{64}, \ldots, k_{127}$ to model the second half of the 128-bit key and the upper limit of r is increased to 48, which results in an overall number of 9472 indeterminates.

AddConstants (AC): This operation forms a matrix from a round constant consisting of a tuple of six bits $(b_5, b_4, b_3, b_2, b_1, b_0)$ (see Table 1 for the concrete values) and a tuple of eight bits $(a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0)$ which encodes the binary representation of the key size of LED. The matrix formed in this way (see Section 2.2.2) is added to the state using bitwise XOR.

To represent this operation by polynomials, we distinguish whether a key addition precedes AC or not and whether we model LED-64 or LED-128. We start with the

$$
\begin{array}{ll}
x_i^{(r)} = z_i^{(r-1)} + a_7 & \text{for } i \in \{0, 16\} \\
x_i^{(r)} = z_i^{(r-1)} + a_6 & \text{for } i \in \{1, 17\} \\
x_i^{(r)} = z_i^{(r-1)} + a_5 & \text{for } i \in \{2, 18\} \\
x_i^{(r)} = z_i^{(r-1)} + a_4 & \text{for } i \in \{3\} \\
x_i^{(r)} = z_i^{(r-1)} + a_4 + 1 & \text{for } i \in \{19\} \\
x_i^{(r)} = z_i^{(r-1)} + a_3 & \text{for } i \in \{32, 48\} \\
x_i^{(r)} = z_i^{(r-1)} + a_2 & \text{for } i \in \{33, 49\} \\
x_i^{(r)} = z_i^{(r-1)} + a_1 + 1 & \text{for } i \in \{34, 50\} \\
x_i^{(r)} = z_i^{(r-1)} + a_0 & \text{for } i \in \{35\} \\
x_i^{(r)} = z_i^{(r-1)} + a_0 + 1 & \text{for } i \in \{51\}
\end{array}
\qquad
\begin{array}{ll}
x_i^{(r)} = z_i^{(r-1)} + 1 & \text{for } i \in \{19, 34, 50, 51\} \\
x_i^{(r)} = z_i^{(r-1)} + b_5^{(r)} & \text{for } i \in \{5, 37\} \\
x_i^{(r)} = z_i^{(r-1)} + b_4^{(r)} & \text{for } i \in \{6, 38\} \\
x_i^{(r)} = z_i^{(r-1)} + b_3^{(r)} & \text{for } i \in \{7, 39\} \\
x_i^{(r)} = z_i^{(r-1)} + b_2^{(r)} & \text{for } i \in \{21, 53\} \\
x_i^{(r)} = z_i^{(r-1)} + b_1^{(r)} & \text{for } i \in \{22, 54\} \\
x_i^{(r)} = z_i^{(r-1)} + b_0^{(r)} & \text{for } i \in \{23, 55\} \\
x_i^{(r)} = z_i^{(r-1)} & \text{otherwise}
\end{array}
$$

Figure 26: Polynomials modelling the AC operation of LED.

simplest case, i.e. modelling the operation without a key addition, where we get equations as shown in Figure 26.

Every fourth round AC is preceded with a key addition which also includes input whitening. In case of LED-64, we simply add the variable k_i to the ith equation above. For LED-128, the key variables $k_0, \ldots, k_{63}$ and $k_{64}, \ldots, k_{127}$ have to be used in an alternating way according to the key usage, as shown in Figure 8. Further note that for $r = 1$ the variables $z_i^{(r-1)}$ have to be replaced by the plaintext variables p_i.

SubCells (SC): During this step, every entry x of the state matrix is replaced by the element $S[x]$ from the S-box given in Table 2. Let $x_0 \parallel x_1 \parallel x_2 \parallel x_3$ be the 4-bit sized input and $y_0 \parallel y_1 \parallel y_2 \parallel y_3$ the 4-bit sized output of S. An easy interpolation computation shows that the S-box can be represented by polynomials as illustrated in Figure 27. In our polynomial representation of the LED encryption algorithm, this step will be combined with the next.

$$
\begin{aligned}
y_0 &= x_0x_1x_3 + x_0x_2x_3 + x_1x_2x_3 + x_1x_2 + x_0 + x_2 + x_3 + 1 \\
y_1 &= x_0x_1x_3 + x_0x_2x_3 + x_0x_2 + x_0x_3 + x_2x_3 + x_0 + x_1 + 1 \\
y_2 &= x_0x_1x_3 + x_0x_2x_3 + x_1x_2x_3 + x_0x_1 + x_0x_2 + x_0 + x_2 \\
y_3 &= x_1x_2 + x_0 + x_1 + x_3
\end{aligned}
$$

Figure 27: Polynomials modelling the S-box of LED.

ShiftRows (SR): For $i = 1, 2, 3, 4$, the ith row of the state matrix is shifted cyclically to the left by $i - 1$ positions. Equivalently, this permutation of the 64-bit state in cyclic

notation can be described by

$$\begin{aligned}\sigma = &(16\ 28\ 24\ 20)(17\ 29\ 25\ 21)(18\ 30\ 26\ 22)(19\ 31\ 27\ 23)\\ &(32\ 40)(33\ 41)(34\ 42)(35\ 43)(36\ 44)(37\ 45)(38\ 46)(39\ 47)\\ &(48\ 52\ 56\ 60)(49\ 53\ 57\ 61)(50\ 54\ 58\ 62)(51\ 55\ 59\ 63)\ .\end{aligned}$$

Note that the first row of the state matrix stays fixed under the SR permutation. Thus, the indices $0, \ldots, 15$ do not appear in the above representation of σ.

Now we model the combined effect of SC and SR. Let $i_1 = 4i$, $i_2 = 4i+1$, $i_3 = 4i+2$, and $i_4 = 4i+3$ for $i = 0, \ldots, 15$. Then, in round r, we get the equations, as shown in Figure 28.

$$\begin{aligned}y^{(r)}_{\sigma(i_1)} &= x^{(r)}_{i_1}x^{(r)}_{i_2}x^{(r)}_{i_4} + x^{(r)}_{i_1}x^{(r)}_{i_3}x^{(r)}_{i_4} + x^{(r)}_{i_2}x^{(r)}_{i_3}x^{(r)}_{i_4} +\\ &\quad x^{(r)}_{i_2}x^{(r)}_{i_3} + x^{(r)}_{i_1} + x^{(r)}_{i_3} + x^{(r)}_{i_4} + 1\\ y^{(r)}_{\sigma(i_2)} &= x^{(r)}_{i_1}x^{(r)}_{i_2}x^{(r)}_{i_4} + x^{(r)}_{i_1}x^{(r)}_{i_3}x^{(r)}_{i_4} + x^{(r)}_{i_1}x^{(r)}_{i_3} +\\ &\quad x^{(r)}_{i_1}x^{(r)}_{i_4} + x^{(r)}_{i_3}x^{(r)}_{i_4} + x^{(r)}_{i_1} + x^{(r)}_{i_2} + 1\\ y^{(r)}_{\sigma(i_3)} &= x^{(r)}_{i_1}x^{(r)}_{i_2}x^{(r)}_{i_4} + x^{(r)}_{i_1}x^{(r)}_{i_3}x^{(r)}_{i_4} + x^{(r)}_{i_2}x^{(r)}_{i_3}x^{(r)}_{i_4} +\\ &\quad x^{(r)}_{i_1}x^{(r)}_{i_2} + x^{(r)}_{i_1}x^{(r)}_{i_3} + x^{(r)}_{i_1} + x^{(r)}_{i_3}\\ y^{(r)}_{\sigma(i_4)} &= x^{(r)}_{i_2}x^{(r)}_{i_3} + x^{(r)}_{i_1} + x^{(r)}_{i_2} + x^{(r)}_{i_4}\end{aligned}$$

Figure 28: Polynomials modelling the combined SC and SR operations of LED.

MixColumnsSerial (MCS): Every column v of the state matrix is replaced by the product $M \cdot v$, where M is the matrix

$$M = \begin{pmatrix} 4 & 1 & 2 & 2\\ 8 & 6 & 5 & 6\\ \mathtt{B} & \mathtt{E} & \mathtt{A} & 9\\ 2 & 2 & \mathtt{F} & \mathtt{B}\end{pmatrix}\ .$$

Let $y^{(r)}_0 \parallel \cdots \parallel y^{(r)}_{63}$ be the state of the cipher after SR has been executed in round r and let $z^{(r)}_0 \parallel \cdots \parallel z^{(r)}_{63}$ be its state after MCS. The entries of the state matrix are the field elements $y^{(r)}_{4i}x^3 + y^{(r)}_{4i+1}x^2 + y^{(r)}_{4i+2}x + y^{(r)}_{4i+3}$ of $\mathbb{F}_{16}$. Plugging these into the above matrix

multiplication yields, for instance, the following first entry of the resulting state matrix:

$$\begin{aligned} z_0^{(r)}x^3 + z_1^{(r)}x^2 + z_2^{(r)}x + z_3^{(r)} = \; & x^2 \cdot (y_0^{(r)}x^3 + y_1^{(r)}x^2 + y_2^{(r)}x + y_3^{(r)}) \\ + \; & 1 \cdot (y_{16}^{(r)}x^3 + y_{17}^{(r)}x^2 + y_{18}^{(r)}x + y_{19}^{(r)}) \\ + \; & x \cdot (y_{32}^{(r)}x^3 + y_{33}^{(r)}x^2 + y_{34}^{(r)}x + y_{35}^{(r)}) \\ + \; & x \cdot (y_{48}^{(r)}x^3 + y_{49}^{(r)}x^2 + y_{50}^{(r)}x + y_{51}^{(r)}) \; . \end{aligned}$$

After expanding, simplifying, and comparing the coefficients of $1, x, x^2, x^3$, we finally get 64 equations, listed in Figure 29, where $j_k = 4i + k$, $j_{k+4} = 4i + 16 + k$, $j_{k+8} = 4i + 32 + k$, and $j_{k+12} = 4i + 48 + k$ for $i, k \in \{0, 1, 2, 3\}$.

$$\begin{aligned}
z_{j_0}^{(r)} &= y_{j_2}^{(r)} + y_{j_4}^{(r)} + y_{j_9}^{(r)} + y_{j_{13}}^{(r)} \\
z_{j_1}^{(r)} &= y_{j_0}^{(r)} + y_{j_3}^{(r)} + y_{j_5}^{(r)} + y_{j_{10}}^{(r)} + y_{j_{14}}^{(r)} \\
z_{j_2}^{(r)} &= y_{j_0}^{(r)} + y_{j_1}^{(r)} + y_{j_6}^{(r)} + y_{j_8}^{(r)} + y_{j_{11}}^{(r)} + y_{j_{12}}^{(r)} + y_{j_{15}}^{(r)} \\
z_{j_3}^{(r)} &= y_{j_1}^{(r)} + y_{j_7}^{(r)} + y_{j_8}^{(r)} + y_{j_{12}}^{(r)} \\
z_{j_4}^{(r)} &= y_{j_0}^{(r)} + y_{j_3}^{(r)} + y_{j_5}^{(r)} + y_{j_6}^{(r)} + y_{j_8}^{(r)} + y_{j_{10}}^{(r)} + y_{j_{13}}^{(r)} + y_{j_{14}}^{(r)} \\
z_{j_5}^{(r)} &= y_{j_0}^{(r)} + y_{j_1}^{(r)} + y_{j_4}^{(r)} + y_{j_6}^{(r)} + y_{j_7}^{(r)} + y_{j_8}^{(r)} + y_{j_9}^{(r)} + y_{j_{11}}^{(r)} + y_{j_{12}}^{(r)} + y_{j_{14}}^{(r)} + y_{j_{15}}^{(r)} \\
z_{j_6}^{(r)} &= y_{j_1}^{(r)} + y_{j_2}^{(r)} + y_{j_5}^{(r)} + y_{j_7}^{(r)} + y_{j_8}^{(r)} + y_{j_9}^{(r)} + y_{j_{10}}^{(r)} + y_{j_{13}}^{(r)} + y_{j_{15}}^{(r)} \\
z_{j_7}^{(r)} &= y_{j_2}^{(r)} + y_{j_4}^{(r)} + y_{j_5}^{(r)} + y_{j_9}^{(r)} + y_{j_{11}}^{(r)} + y_{j_{12}}^{(r)} + y_{j_{13}}^{(r)} \\
z_{j_8}^{(r)} &= y_{j_1}^{(r)} + y_{j_3}^{(r)} + y_{j_4}^{(r)} + y_{j_5}^{(r)} + y_{j_6}^{(r)} + y_{j_7}^{(r)} + y_{j_8}^{(r)} + y_{j_9}^{(r)} + y_{j_{11}}^{(r)} + y_{j_{15}}^{(r)} \\
z_{j_9}^{(r)} &= y_{j_0}^{(r)} + y_{j_2}^{(r)} + y_{j_5}^{(r)} + y_{j_6}^{(r)} + y_{j_7}^{(r)} + y_{j_8}^{(r)} + y_{j_9}^{(r)} + y_{j_{10}}^{(r)} + y_{j_{12}}^{(r)} \\
z_{j_{10}}^{(r)} &= y_{j_0}^{(r)} + y_{j_1}^{(r)} + y_{j_3}^{(r)} + y_{j_6}^{(r)} + y_{j_7}^{(r)} + y_{j_8}^{(r)} + y_{j_9}^{(r)} + y_{j_{10}}^{(r)} + y_{j_{11}}^{(r)} + y_{j_{13}}^{(r)} \\
z_{j_{11}}^{(r)} &= y_{j_0}^{(r)} + y_{j_2}^{(r)} + y_{j_3}^{(r)} + y_{j_4}^{(r)} + y_{j_5}^{(r)} + y_{j_6}^{(r)} + y_{j_8}^{(r)} + y_{j_{10}}^{(r)} + y_{j_{14}}^{(r)} + Y_{j_{15}}^{(r)} \\
z_{j_{12}}^{(r)} &= y_{j_1}^{(r)} + y_{j_5}^{(r)} + y_{j_9}^{(r)} + y_{j_{10}}^{(r)} + y_{j_{11}}^{(r)} + y_{j_{13}}^{(r)} + y_{j_{15}}^{(r)} \\
z_{j_{13}}^{(r)} &= y_{j_2}^{(r)} + y_{j_6}^{(r)} + y_{j_{10}}^{(r)} + y_{j_{11}}^{(r)} + y_{j_{12}}^{(r)} + y_{j_{14}}^{(r)} \\
z_{j_{14}}^{(r)} &= y_{j_0}^{(r)} + y_{j_3}^{(r)} + y_{j_4}^{(r)} + y_{j_7}^{(r)} + y_{j_{11}}^{(r)} + y_{j_{12}}^{(r)} + y_{j_{13}}^{(r)} + y_{j_{15}}^{(r)} \\
z_{j_{15}}^{(r)} &= y_{j_0}^{(r)} + y_{j_4}^{(r)} + y_{j_8}^{(r)} + y_{j_9}^{(r)} + y_{j_{10}}^{(r)} + y_{j_{11}}^{(r)} + y_{j_{12}}^{(r)} + y_{j_{14}}^{(r)} + y_{j_{15}}^{(r)}
\end{aligned}$$

Figure 29: Polynomials modelling the MCS operation of LED.

The final key addition, which also finishes the algebraic representation of the LED-64

block cipher, is described by the equations

$$c_i = z_i^{(32)} + k_i$$

for $i = 0, \ldots, 63$. It is clear that LED-128 has a similar description, using additional indeterminates for the second key and the extra rounds.

2.6.2 Algebraic Representation of the LED Fault Equations

The algebraic representation of LED-64 constructed above is not suitable to launch a successful algebraic attack. It involves too many non-linear equations in too many indeterminates. To reconstruct the secret key from given (correct or faulty) plaintext – ciphertext pairs requires additional information. This information will be furnished by a fault attack. In Section 2.4, we discussed a method for injecting faults and using it to break LED-64 by exhaustive search. In the following, we construct a polynomial version of the fault equations, as shown in Figure 13.

Since these equations involve the inverse S-box map $S^{-1} : \mathbb{F}_{16} \longrightarrow \mathbb{F}_{16}$, we need to find a polynomial representation of this map. Using the values of this map, as given in Table 5, and univariate interpolation, we construct the following polynomial representation of S^{-1}:

$$\begin{aligned} S^{-1}(y) = {} & (x^2+1) + (x^2+1)y + (x^3+x)y^2 + (x^3+x^2+1)y^3 + xy^4 + \\ & (x^3+1)y^5 + (x^3+1)y^7 + (x+1)y^9 + (x^2+1)y^{10} + (x^3+1)y^{11} + \\ & (x^3+x)y^{12} + (x+1)y^{13} + (x^3+x^2+1)y^{14} \ . \end{aligned}$$

Next, we plug the right-hand sides of the fault equations into this polynomial. We get 16 polynomial fault equations which are defined over the polynomial ring

$$\mathbb{F}_{16}[a, b, c, d, \bar{k}_0, \ldots, \bar{k}_{15}, \bar{c}_0, \ldots, \bar{c}_{15}, \bar{c}'_0, \ldots, \bar{c}'_{15}] \ .$$

For every group of equations $E_{t,0}, E_{t,1}, E_{t,2}, E_{t,3}$ having the same left-hand side $t \in \{a, b, c, d\}$, we can form three differences $E_{t,0} - E_{t,i} = 0$ with $i = 1, 2, 3$. Now, comparing coefficients for $\{1, x, x^2, x^3\}$ yields 48 equations in the bits $k_0, \ldots, k_{63}$ of the secret key, the bits $c_0, \ldots, c_{63}$ of the correct ciphertext, and the bits $c'_0, \ldots, c'_{63}$ of the faulty ciphertext. Notice that we can use the field equations $k_i^2 + k_i = 0$, $c_i^2 + c_i = 0$, and $(c'_i)^2 + c'_i = 0$ for simplification here.

Altogether, we find 48 polynomials in $\mathbb{F}_2[k_0, \ldots, k_{63}, c_0, \ldots, c_{63}, c'_0, \ldots, c'_{63}]$. They all have degree 3 and are composed of 3400 – 8800 terms. These polynomials will be called the *fault polynomials*.

2.6.3 Experimental Results

In the preceding two sections, we derived polynomials describing the encryption map of LED-64 and additional information gained from a fault attack. All in all, we found 6208

polynomials in 6336 indeterminates describing the encryption map, 6336 field equations, and 48 fault polynomials in 192 indeterminates.

As mentioned previously, we assume that we are able to mount a known-plaintext-attack and to repeat encryption involving the same key and the fault injection described previously. For every concrete instance of this attack, we can therefore substitute the plaintext bits, correct ciphertext bits, and faulty ciphertext bits into our polynomials. After this substitution, we have 6208 polynomials in 6208 indeterminates for the encryption map, 6208 field equations, and 48 fault polynomials in the 64 indeterminates of the secret key.

The resulting fault polynomials consist typically of $40 - 150$ terms. Some of them (usually no more than 5) drop their degree and become linear. Of course, these linear polynomials are particularly valuable, since they decrease the complexity of the problem by one dimension. In the experiments reported below, it turned out to be beneficial to interreduce the fault polynomials after substitution in order to generate more linear ones.

The polynomial systems can be solved using various techniques. For our experiments, we applied the algorithms for conversion to a SAT-solving problem explained in [137].

All experiments were performed on a workstation having eight 3.5 GHz Xeon cores and 50 GB of RAM. We used the SAT-solvers Minisat 2.2 (MS) and CryptoMiniSat 2.9.4 (CMS). All timings are averages over ten LED-64 instances with random plaintext, key, and fault values.

In our first experiments, we measured the time to solve a given polynomial system without further modifications using SAT-solvers, see the first part of Table 9. For the second set of experiments, we first interreduced the fault polynomials and then additionally appended all linear polynomials obtained this way to the system. In some cases we were able to find more linear dependencies between the key indeterminates, thereby reducing the dimension even further. Moreover, the SAT-solvers appear to benefit from this simplification, because it is typically the number of terms in a polynomial that complicates its logical representation. This seemingly minor modification results in a meaningful speed-up, as we can see in the second part of Table 9.

In summary, it is clear that the proposed fault attack is able to break the LED-64 encryption scheme. While it is slower than the direct fault attack presented in Section 2.4, it does not rely on the specific properties underlying the key filtering steps there and it offers numerous possibilities for optimization.

2.7 Conclusion

In this chapter, we presented multiple contributions to the field of fault analysis of lightweight block ciphers.

First, we demonstrated that the LED-64 block cipher has a vulnerability to fault-based attacks which roughly matches that of AES. The improved protection mechanism of

Table 9: Average SAT-solver timings.

SAT solver	MS (1 thread)	CMS (1 thread)	CMS (4 threads)
time (in sec)	90,852	71,656	22,639
time (in h)	25.23	19.90	6.28

(a) Standard.

SAT solver	MS (1 thread)	CMS (1 thread)	CMS (4 threads)
time (in sec)	36,665	52,835	11,829
time (in h)	10.18	14.67	3.28

(b) With additional linear equations.

LED can be overcome through a filtering process of subsets of key candidates using fault equations. Furthermore, we discussed extensions of single-nibble to diagonal fault injections, which allow faults in up to four nibbles, without affecting the subsequent cryptanalysis. Obviously, this gives an attacker a higher degree of freedom when injecting faults. In our experiments, we could show that in most cases already one fault injection is enough to break LED-64, i.e. to reconstruct the entire 64-bit key.

The 128-bit key version of LED is more challenging to attack. A direct application of the techniques used on LED-64 is not sufficient to mount a successful attack on LED-128, since the latter employs two subkeys, which are independent from each other, i.e. which are not connected through a key schedule. This is in contrast to AES, where the knowledge of one subkey allows the reconstruction of the master key, due to the bijectivity of the AES key schedule. Although LED-128's strength collapses if an attacker has the ability to set one half of the key bits to a known value (e.g., during the transfer from a secure memory location), this is usually quite a restrictive assumption and assumes a very powerful adversary.

As a second contribution, we introduced the generic concept of multi-stage fault attacks which target individual subkeys by multiple fault injections. One stage consists of several fault injections followed by mathematical analysis that yields a set of candidates for a subkey. We presented an algorithm that balances the number of fault injections allocated to different stages, in order to keep the sizes of the final key candidate sets sufficiently small for brute-force search. The generic algorithm estimates the expected effort for each stage and then decides on the number of fault injections to be performed based on user-specified threshold variables. Next to LED-128, we also illustrated the successful application of the general principle to PRINCE, another lightweight block cipher, which employs (almost) independent 64-bit subkeys. This approach allows us to break both

schemes with 3 to 4 fault injections on average. Similarly to the case of LED-64, we also discussed diagonal fault injections for LED-128 and PRINCE.

The third contribution of this chapter is the extension of the fault attack on LED-64 to an algebraic setting. After providing a complete algebraic description of the LED block cipher and showing how to convert the previously introduced fault equations into fault polynomials, it turned out that the combined polynomial system was solvable by state-of-the-art SAT solvers. Thus, algebraic attacks augmented with information from fault injections are able to break the LED-64 encryption algorithm in practice. Extending algebraic fault attacks to other ciphers, such as LED-128 and PRINCE, might be an interesting next step. Due to the larger key size of 128 bits and the necessity to use multiple fault injections, these ciphers are much more challenging to analyse, just as in the of the classical differential (multi-stage) fault attack scenario.

Chapter 3

Fault-based Attacks on the Bel-T Block Cipher Family

3.1 Introduction

In this chapter we investigate the vulnerability of the block cipher family Bel-T to fault-based attacks. Bel-T has been approved as a standard of Republic of Belarus in 2011. The specification of Bel-T in Russian language is available from the web site of its developer, the Research Institute for Applied Problems of Mathematics and Informatics of the Belarussian State University [98]. We are not aware of an English-language specification of this cipher, nor of any published results on its security. Bel-T follows the Lai-Massey scheme [176] which has similarities with both substitution-permutation networks and Feistel networks. Earlier ciphers constructed according to this scheme include IDEA [176] and its extension IDEA NXT, also known as FOX [144]. Fault-based attacks are known for both IDEA [87] and FOX [79], yet these attacks are cipher-specific and do not utilize the Lai-Massey construction *per se* and hence are not applicable to Bel-T. There are three versions of Bel-T, which use secret keys of different lengths. Our fault-based attack is applicable to all three versions but requires a different number of fault injections in order to recover the entire secret key. The attack utilizes the property of Bel-T that the same functionality with reordered parts of the secret key is used for encryption and decryption.

The differential fault analysis of Bel-T was presented at DATE 2015 [143].

Outline. The remainder of the chapter is organized as follows. Section 3.2 provides the specification of the Bel-T block cipher family which is the first English-language description of Bel-T as far as we know. In Section 3.3, the new fault-based attack is described together with the results of our comprehensive simulation-based experiments. The requirements on the fault injection precision and the countermeasures against the attack are discussed in Section 3.4. Section 3.5 concludes the chapter.

3.2 The Block Cipher Bel-T

We describe the specification of Bel-T using the same notation as in the original (Russian-language) document [98]. In particular, given $u, v \in \{0,1\}^n$, $u \oplus v$ stands for the bit-wise addition modulo 2 (exclusive-or) of u and v, and $u \boxplus v$ and $u \boxminus v$, respectively, stand for the arithmetical addition and subtraction of u and v modulo 2^n, where u and v are interpreted as unsigned integers.

According to that document, Bel-T is foreseen for use in six modes: electronic codebook (ECB), cipher block chaining (CBC), cipher feedback (CFB), counter (CTR), message authentication code (MAC), and hashing. Since our attack targets the block cipher itself, we do not discuss these six modes in detail.

Bel-T is a block cipher which encrypts a 128-bit plaintext X using a 256-bit value $\theta = \theta_1 \parallel \ldots \parallel \theta_8$ with 32-bit words θ_i for $1 \leq i \leq 8$, to obtain the 128-bit ciphertext Y. The Bel-T family consists of three ciphers which employ secret keys of different lengths (128 bits, 192 bits, and 256 bits) and are identical otherwise. We call these versions Bel-T-128, Bel-T-192, and Bel-T-256, respectively (the original document does not use explicit names for these versions). The key setup is as follows:

- Bel-T-256: The value θ is identical to the 256-bit secret key.
- Bel-T-192: The first six words $\theta_1, \ldots, \theta_6$ of θ correspond to the 192-bit secret key and the remaining two words θ_7 and θ_8 are obtained by computing $\theta_7 := \theta_1 \oplus \theta_2 \oplus \theta_3$ and $\theta_8 := \theta_4 \oplus \theta_5 \oplus \theta_6$.
- Bel-T-128: The first four words $\theta_1, \ldots, \theta_4$ of θ correspond to the 128-bit secret key and the remaining four words $\theta_5, \ldots, \theta_8$ are obtained by computing $\theta_5 := \theta_1$, $\theta_6 := \theta_2$, $\theta_7 := \theta_3$ and $\theta_8 := \theta_4$.

The encryption is written by $Y = \mathcal{E}_\theta(X)$, the decryption is written by $Y = \mathcal{D}_\theta(X)$. Both encryption and decryption are organised in eight rounds. Refer to Algorithm 4 (belt_encrypt) and Algorithm 5 (belt_decrypt) for the respective pseudocodes.

The rounds use different sets of *round keys* and are identical otherwise. Round keys are 32-bit values $K_1, \ldots, K_{56}$, where $K_1 = \theta_1$, $K_2 = \theta_2, \ldots, K_8 = \theta_8$, $K_9 = \theta_1, \ldots, K_{56} = \theta_8$. In round $i \in \{1, \ldots, 8\}$, seven round keys K_{7i-j}, with $0 \leq j \leq 6$, are used. Their order is shown in Table 11, for encryption from top to bottom and for decryption the other way round. Also note the different ordering of the K_{7i-j} during encryption (top) and decryption (bottom). The method setup_keys in Algorithms 4 and 5 loads the keys as discussed above into the array variable K.

Three mappings G_5, G_{13}, and $G_{21} : \mathbb{F}_2^{32} \to \mathbb{F}_2^{32}$ are used, where G_r maps a 32-bit word $u = u_1 \parallel u_2 \parallel u_3 \parallel u_4$, with $u_i \in \mathbb{F}_2^8$, as follows:

$$G_r(u) = (H(u_1) \parallel H(u_2) \parallel H(u_3) \parallel H(u_4)) \lll r \ .$$

Algorithm 4: belt_encrypt(θ, X)

Inputs:
$\theta \in \mathbb{F}_2^{256}, X \in \mathbb{F}_2^{128}$
Outputs:
$Y \in \mathbb{F}_2^{128}$
Algorithm:
1. $K \leftarrow \mathsf{setup_keys}(\theta)$
2. $a \parallel b \parallel c \parallel d \leftarrow X$
3. **for** $i \in \{1, \ldots, 8\}$ **do**
4. $\quad b \leftarrow b \oplus G_5(a \boxplus K_{7i-6})$
5. $\quad c \leftarrow c \oplus G_{21}(d \boxplus K_{7i-5})$
6. $\quad a \leftarrow a \boxminus G_{13}(b \boxplus K_{7i-4})$
7. $\quad e \leftarrow G_{21}(b \boxplus c \boxplus K_{7i-3}) \oplus \langle i \rangle_{32}$
8. $\quad b \leftarrow b \boxplus e$
9. $\quad c \leftarrow c \boxminus e$
10. $\quad d \leftarrow d \boxplus G_{13}(c \boxplus K_{7i-2})$
11. $\quad b \leftarrow b \oplus G_{21}(a \boxplus K_{7i-1})$
12. $\quad c \leftarrow c \oplus G_5(d \boxplus K_{7i})$
13. $\quad$ swap a and b
14. $\quad$ swap c and d
15. $\quad$ swap b and c
16. **end**
17. $Y \leftarrow b \parallel d \parallel a \parallel c$
18. **return** Y

Algorithm 5: belt_decrypt(θ, X)

Inputs:
$\theta \in \mathbb{F}_2^{256}, X \in \mathbb{F}_2^{128}$
Outputs:
$Y \in \mathbb{F}_2^{128}$
Algorithm:
1. $K \leftarrow \mathsf{setup_keys}(\theta)$
2. $a \parallel b \parallel c \parallel d \leftarrow X$
3. **for** $i \in \{1, \ldots, 8\}$ **do**
4. $\quad b \leftarrow b \oplus G_5(a \boxplus K_{7i})$
5. $\quad c \leftarrow c \oplus G_{21}(d \boxplus K_{7i-1})$
6. $\quad a \leftarrow a \boxminus G_{13}(b \boxplus K_{7i-2})$
7. $\quad e \leftarrow G_{21}(b \boxplus c \boxplus K_{7i-3}) \oplus \langle i \rangle_{32}$
8. $\quad b \leftarrow b \boxplus e$
9. $\quad c \leftarrow c \boxminus e$
10. $\quad d \leftarrow d \boxplus G_{13}(c \boxplus K_{7i-4})$
11. $\quad b \leftarrow b \oplus G_{21}(a \boxplus K_{7i-5})$
12. $\quad c \leftarrow c \oplus G_5(d \boxplus K_{7i-6})$
13. $\quad$ swap a and b
14. $\quad$ swap c and d
15. $\quad$ swap b and c
16. **end**
17. $Y \leftarrow b \parallel d \parallel a \parallel c$
18. **return** Y

Here, H is the S-box specified in Table 12 and $\lll r$ stands for a cyclical shift to the left by r positions. The value $\langle i \rangle_{32}$ in line 7) of Algorithm 4 stands for the binary number of the round. The diagram in Figure 30 shows the functionality of a round. It also indicates how Bel-T can be implemented in hardware.

Decryption, see Algorithm 5, is identical to encryption with one exception: round key K_{7i} is used in line 4) instead of K_{7i-6}; round key K_{7i-1} is used in line 5) instead of K_{7i-5}; and so forth. This reduces the complexity of the algorithm, in particular for a hardware implementation, where the same circuitry can be used for both encryption and decryption. However, this feature is also instrumental for the fault-based attack presented in the next section.

3.3 Fault Attacks on Bel-T

As has been mentioned above, the fault-based attack scenario assumes that the attacker has physical access to the device that performs the encryption, is capable to encrypt plaintexts of his choice and observe the resulting ciphertexts, to decrypt ciphertexts of his choice and observe the resulting plaintexts, and to inject faults during encryption or decryption. The objective of the attacker is to recover the secret key $\theta = \theta_1 \parallel \ldots \parallel \theta_8$ (recall that $\theta_i \in \{0,1\}^{32}$). The key is stored within the device without being directly

Table 11: Key Usage in Bel-T.

	i	K_{7i-6}	K_{7i-5}	K_{7i-4}	K_{7i-3}	K_{7i-2}	K_{7i-1}	K_{7i}	
Encryption ↓	1	θ_1	θ_2	θ_3	θ_4	θ_5	θ_6	θ_7	↑ Decryption
	2	θ_8	θ_1	θ_2	θ_3	θ_4	θ_5	θ_6	
	3	θ_7	θ_8	θ_1	θ_2	θ_3	θ_4	θ_5	
	4	θ_6	θ_7	θ_8	θ_1	θ_2	θ_3	θ_4	
	5	θ_5	θ_6	θ_7	θ_8	θ_1	θ_2	θ_3	
	6	θ_4	θ_5	θ_6	θ_7	θ_8	θ_1	θ_2	
	7	θ_3	θ_4	θ_5	θ_6	θ_7	θ_8	θ_1	
	8	θ_2	θ_3	θ_4	θ_5	θ_6	θ_7	θ_8	
	i	K_{7i}	K_{7i-1}	K_{7i-2}	K_{7i-3}	K_{7i-4}	K_{7i-5}	K_{7i-6}	

accessible to the attacker.

Let $X \in \{0,1\}^{128}$ be an arbitrary plaintext and let $Y \in \{0,1\}^{128}$ be the ciphertext calculated by the device in absence of any fault injections: $Y = \mathcal{E}_\theta(X)$. The fault-based attack is conducted by series of fault injections, where "fault injection" refers to performing the encryption of the same plaintext X while injecting a transient fault and recording the faulty ciphertext Y^f. It is assumed that the secret key does not change during the whole attack, that is, all fault-affected encryptions are performed using the same θ (and the same X) as the fault-free encryption. Each fault injection determines several bits of θ. Moreover, the same principle is applied to decryptions: Let $\tilde{X}$ be an arbitrary ciphertext and let $\tilde{Y} = \mathcal{D}_\theta(\tilde{X})$ be the plaintext obtained by decryption in absence of faults. Injecting faults during decryption results in deviating plaintexts $\tilde{Y} = \mathcal{D}_\theta^f(\tilde{X})$. Note that ciphertext $\tilde{X}$ does not need to match plaintext X used when performing fault injections during encryption, but the secret key θ is assumed to be identical during all fault-free and fault-affected encryptions and decryptions.

All faults used in our attack are applied during the last of the eight rounds of Bel-T. Before we describe the concrete attacks on Bel-T, we introduce the two fault models that are used in our analysis:

- The random fault model (RFM) assumes that an attacker can inject faults into an element of the state at a freely chosen position, such that its value switches to a random, unknown value.
- The chosen fault model (CFM) assumes that an attacker can inject faults into an element of the state at a freely chosen position, such that the value of the state element switches to a known value.

To perform our attack on Bel-T-128, we require only 4 RFM faults. For Bel-T-192 and Bel-T-256 we likewise need 4 RFM faults, but 3 respectively 6 additional CFM faults. We first describe the attack on Bel-T-128, because it also forms the basis for the fault attacks on Bel-T-192 and Bel-T-256.

Table 12: The Bel-T S-box H.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	B1	94	BA	C8	0A	08	F5	3B	36	6D	00	8E	58	4A	5D	E4
1	85	04	FA	9D	1B	B6	C7	AC	25	2E	72	C2	02	FD	CE	0D
2	5B	E3	D6	12	17	B9	61	81	FE	67	86	AD	71	6B	89	0B
3	5C	B0	C0	FF	33	C3	56	B8	35	C4	05	AE	D8	E0	7F	99
4	E1	2B	DC	1A	E2	82	57	EC	70	3F	CC	F0	95	EE	8D	F1
5	C1	AB	76	38	9F	E6	78	CA	F7	C6	F8	60	D5	BB	9C	4F
6	F3	3C	65	7B	63	7C	30	6A	DD	4E	A7	79	9E	B2	3D	31
7	3E	98	B5	6E	27	D3	BC	CF	59	1E	18	1F	4C	5A	B7	93
8	E9	DE	E7	2C	8F	0C	0F	A6	2D	DB	49	F4	6F	73	96	47
9	06	07	53	16	ED	24	7A	37	39	CB	A3	83	03	A9	8B	F6
A	92	BD	9B	1C	E5	D1	41	01	54	45	FB	C9	5E	4D	0E	F2
B	68	20	80	AA	22	7D	64	2F	26	87	F9	34	90	40	55	11
C	BE	32	97	13	43	FC	9A	48	A0	2A	88	5F	19	4B	09	A1
D	7E	CD	A4	D0	15	44	AF	8C	A5	84	50	BF	66	D2	E8	8A
E	A2	D7	46	52	42	A8	DF	B3	69	74	C5	51	EB	23	29	21
F	D4	EF	D9	B4	3A	62	28	75	91	14	10	EA	77	6C	DA	1D

In the following, we denote the fault-free outputs at the end of encryption and decryption, respectively, by w, x, y, and z, as in Figure 30, and the fault-affected outputs analogously by w', x', y', and z'.

3.3.1 Bel-T-128

We start our attack on Bel-T-128 with the aim to reconstruct subkey K_{7i-1} in the last round of the encryption ($i = 8$), which corresponds to $\theta_7 = \theta_3$. For this purpose, we inject the first fault f_1 (RFM) into the state at position marked L_1 in Figure 30. Note that f_1 can flip an arbitrary number of bits within the 32-bit word and the best results are achieved during filtering if f_1 affects all of the 4 bytes in L_1. The value of f_1 is not immediately known to the attacker. However, it can be derived by observing the correct and faulty ciphertexts Y and Y^{f_1} at positions y and y' of the output of the cipher and calculating the XOR of these values: $f_1 = y \oplus y'$. The fault propagates through the key addition of θ_7, the application of G_{21} and the XOR with the b-part of the state, and creates an XOR-difference $w \oplus w'$, which is also directly observable at the outputs of the cipher. Since the value at L_1 corresponds to y during a fault-free encryption, subkey θ_7 must obey the following formula:

$$G_{21}(L_1 \boxplus \theta_7) \oplus G_{21}((L_1 \oplus f_1) \boxplus \theta_7) = w \oplus w' \ . \tag{3.1}$$

The XOR with the b-part of the state is ignored since it does not influence $w \oplus w'$. All values in the above equation are known except for θ_7. Equation 3.1 is checked for all 2^{32}

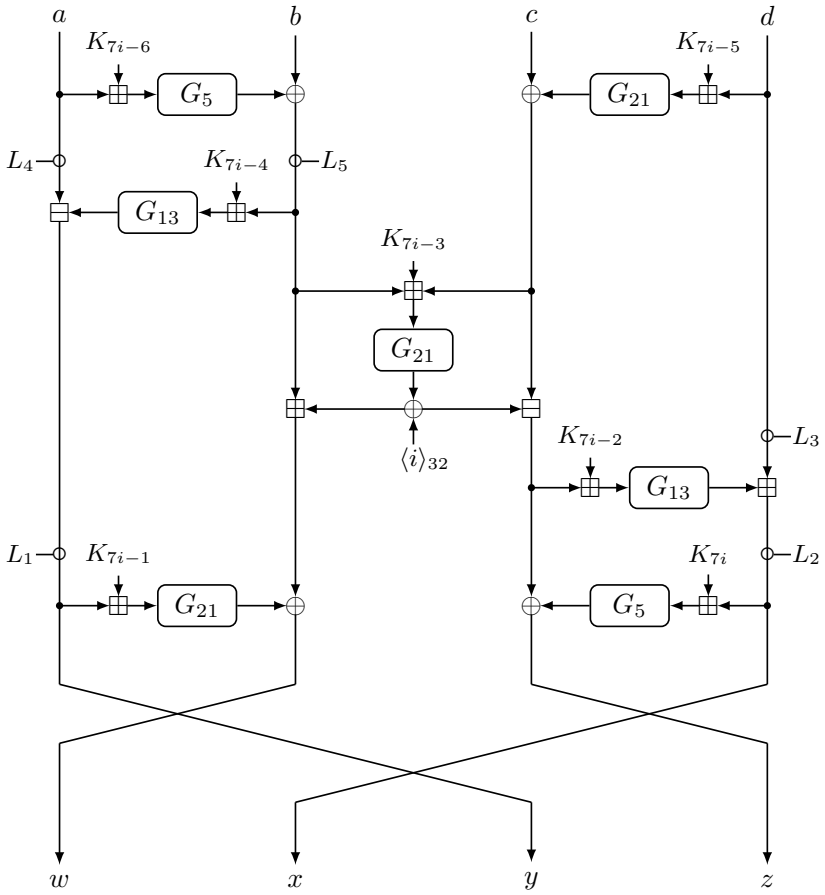


Figure 30: The ith Bel-T round with fault locations L_j.

bit combinations for θ_7, and all candidates that satisfy the equation are collected in a set called Θ_7.

Referring again to Figure 30, we observe that keys added at K_{7i} $(= \theta_8 = \theta_4)$, can be reconstructed analogously. Fault f_2 is injected at position L_2 during encryption in round $i = 8$ and we use the information on $f_2 = x \oplus x'$ and $z \oplus z'$ from the corresponding outputs for our analysis. Thereby, we can reduce the number of candidates for subkey θ_8 and store them in set Θ_8.

For retrieval of the two missing subkeys θ_1 $(= \theta_5)$ and θ_2 $(= \theta_6)$, we exploit the property that encryption and decryption of Bel-T are basically the same. We switch from encryption to decryption and attack again the last round $(i = 1)$. Looking at Table 11, we see that at positions K_{7i} and K_{7i-1} subkeys θ_1 and θ_2 are added to the state. Thus, injecting faults f_3 and f_4 at positions L_1 and L_2 in decryption allows us to reconstruct the two

missing subkeys using the same approach as above. The candidates are stored in sets Θ_1 and Θ_2, respectively.

After the above filtering steps are finished, all of the candidates for the secret key $\theta = \theta_1 \parallel \cdots \parallel \theta_8 = \theta_1 \parallel \theta_2 \parallel \theta_7 \parallel \theta_8 \parallel \theta_1 \parallel \theta_2 \parallel \theta_7 \parallel \theta_8$ must be obviously contained in the following set:

$$\Theta_1 \times \Theta_2 \times \Theta_7 \times \Theta_8 \times \Theta_1 \times \Theta_2 \times \Theta_7 \times \Theta_8 \ .$$

In our experiments, this set was always sufficiently small to perform a brute-force search for the correct key. If not, we can reduce this set by repeating one or all of the above procedures and computing the intersection of the corresponding subkey candidate sets.

3.3.2 Bel-T-192

The attack on Bel-T-192 starts with the same 4 fault injections and the subsequent analysis as in the Bel-T-128-case. Thus, we retrieve information on the values θ_1, θ_2, $\theta_7 = \theta_1 \oplus \theta_2 \oplus \theta_3$, and $\theta_8 = \theta_4 \oplus \theta_5 \oplus \theta_6$ and store them in sets $\Theta_1, \Theta_2, \Theta_7$, and Θ_8. Since $\theta_3 = \theta_7 \oplus \theta_1 \oplus \theta_2$, the set of subkey candidates Θ_3 is obtained from Θ_1, Θ_2 and Θ_7 by collecting all XOR combinations of values from these three sets. The missing subkeys are two out of three from θ_4, θ_5, and θ_6, since we already know the XOR of the latter three from θ_8.

Our next target is the key added at position K_{7i-2} in the last round of encryption, which corresponds to θ_6. The determination of this subkey cannot be achieved by injecting an RFM fault before the key addition of K_{7i-2}, because the difference propagates through the latter and G_{13} but the result is masked by the $\boxplus$-operation with the d-state, which is unknown at this point and therefore leads to an unpredictable outcome. We switch the fault model from RFM to CFM and assume that an attacker can inject faults which set a part of the state to a fixed and known value. We assume for simplicity that this value is 0 which is also reasonable from a practical perspective, since zeroing registers should be comparably easy. However, the analysis works for an arbitrary value. The CFM fault f_5 is injected at position L_3, resetting the d-state at this point to 0. This allows us to build an equation for filtering θ_6-candidates of the form

$$G_{13}(s \boxplus \theta_6) \boxplus 0 = x' \tag{3.2}$$

where $s = G_5(x \boxplus \theta_8) \oplus z$. Recall that we already reduced the number of θ_8-candidates at this point. A search over all combinations of θ_6 and θ_8 candidates is therefore feasible, as long as the number of θ_8 values is not too large. During our experiments this was never the case though, as described later. Finally, we again store all θ_6 candidates which satisfy Equation 3.2 in set Θ_6.

Repeating the above approach for decryption is not necessary, since subkey θ_3 is added at position K_{7i-2}, see Table 11, and we already restricted the number of candidates for the latter in our previous analysis.

Our next target is the key addition K_{7i-4}, where subkey θ_4 is processed during the last round of encryption. We need dual faults f_6 and f_7 at L_4 and L_5 which reset the particular state words to 0 and circumvent masking with unknown state elements a and b. The θ_4 values that satisfy the resulting filtering equation

$$0 \boxminus G_{13}(0 \boxplus \theta_4) = y' \tag{3.3}$$

are stored in set Θ_4.

The last missing subkey θ_5 can be reconstructed from the knowledge of the candidates for θ_4, θ_6 and $\theta_8 = \theta_4 \oplus \theta_5 \oplus \theta_6$. All θ_5 candidates satisfying these relationships are stored in set Θ_5, and the complete key is again found by the brute force search in $\Theta_1 \times \cdots \times \Theta_8$. In summary, 7 faults are sufficient to reconstruct the secret key θ for Bel-T-192.

3.3.3 Bel-T-256

Mounting the Bel-T-128 attack on Bel-T-256 (by injecting 4 RFM faults f_1, f_2, f_3, and f_4) we can reconstruct candidates for subkeys θ_1, θ_2, θ_7, and θ_8. Unlike for Bel-T-192, the remaining subkeys θ_3, θ_4, θ_5, and θ_6 are independent from the reconstructed keys. All following faults are CFMs in the last round of encryption or decryption, which set the attacked element(s) to 0. Using an approach similar to Bel-T-192, we can collect information on subkey θ_6. Repeating the same attack on the last round of decryption, fault f_6 at position L_3 gives us candidates for θ_3. The filtering equations for injections of faults f_5 and f_6 are of a similar shape as Equation 3.2.

The last two subkeys θ_4 and θ_5 are reconstructed by dual fault injections f_7 and f_8, and f_9 and f_{10} at locations L_4 and L_5 during last round of encryption and decryption, respectively, see again Table 11 and Figure 30, with filtering equations similar to Equation 3.3. In summary, 10 fault injections are sufficient to break Bel-T-256.

3.3.4 Experimental Results

We performed 5,000 runs for each of the above attacks and recorded the sizes of the respective (sub)key candidate sets. Table 13 gives an overview on the results of our simulations for Bel-T-128, Bel-T-192 and Bel-T-256. We listed only the results for the actual (sub)keys and omitted the information on those that are generated during the Bel-T "key-schedule", like θ_7 and θ_8 in case of Bel-T-192, since they do not provide any additional insights. Moreover, Figure 31 shows the corresponding distributions for the sizes of the θ-candidate sets. The results clearly show that the fault injections as detailed in Section 3.3 are sufficient to reduce the key space of each Bel-T variant such that a subsequent brute-force on the remaining θ-candidates becomes feasible. The implementation of our attack was written in C/C++, but did not exploit parallelisation. All experiments were performed on a workstation with an AMD Opteron 6172 Processor operating at 2.1 GHz. Under these circumstances we measured, for the analysis of one

instance (i.e. reconstruction of one key) without the subsequent brute-force, average running times of 148.0 (Bel-T-128), 287.0 (Bel-T-192), and 687.0 (Bel-T-256) seconds, respectively.

Table 13: Statistics on the binary logarithms for the number of key candidates.

		Θ	Θ_1	Θ_2	Θ_3	Θ_4	Θ_5	Θ_6	Θ_7	Θ_8
Bel-T-128	min	0.00	0.00	0.00	0.00	0.00	-	-	-	-
	max	22.00	10.00	10.58	17.00	10.58	-	-	-	-
	avg	5.11	3.32	3.17	5.64	3.00	-	-	-	-
	med	4.58	1.00	1.00	1.00	1.00	-	-	-	-
Bel-T-192	min	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-	-
	max	40.00	10.32	10.00	17.58	0.00	19.17	9.58	-	-
	avg	10.06	3.32	3.00	7.71	0.00	11.26	2.81	-	-
	med	9.17	1.00	1.00	3.58	0.00	2.00	1.00	-	-
Bel-T-256	min	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	max	39.00	10.00	10.00	10.00	0.00	0.00	10.58	16.00	10.58
	avg	7.63	3.17	3.17	3.17	0.00	0.00	3.32	4.46	3.32
	med	7.00	1.00	1.00	1.00	0.00	0.00	1.00	1.00	1.00

3.4 Practical Issues and Countermeasures

The described attacks require the capability of the attacker to selectively target individual 32-bit words of the cipher state, or pairs of 32-bit words in the case of Bel-T-192 and Bel-T-256 without inducing faults elsewhere in the cipher. This rules out low-precision fault-injection techniques like clock manipulation, underpowering, overheating or illumination by X-rays. In contrast, high-precision attacks using lasers [103] or electromagnetic EM pulses [101] may require deprocessing and preparation of the device under attack as well as adequate instrumentation. The need to inject CFM faults translates to spatial resolution below gate level, i.e. the ability to target individual transistors within a logic gate or a memory cell.

If Bel-T is implemented in hardware, the attacks may either target the cells of the register in which the respective subkey is stored, or the combinational logic which drives this register. If Bel-T is implemented in software and runs on a microprocessor, the attacks may target the architectural register in which the corresponding variable is stored or the arithmetic-logic unit which calculates it. In both cases, the attacker must determine which register in the circuit or microprocessor stores the value to be manipulated and which point of time corresponds to round eight of encryption or decryption. If the attacker has control over the clock signal of the circuit or microprocessor, the timing of fault injection is simplified; otherwise the attacker must deduce the right point in time from side-channel analysis, e.g. by observing variations in power consumption at the beginning of each round.

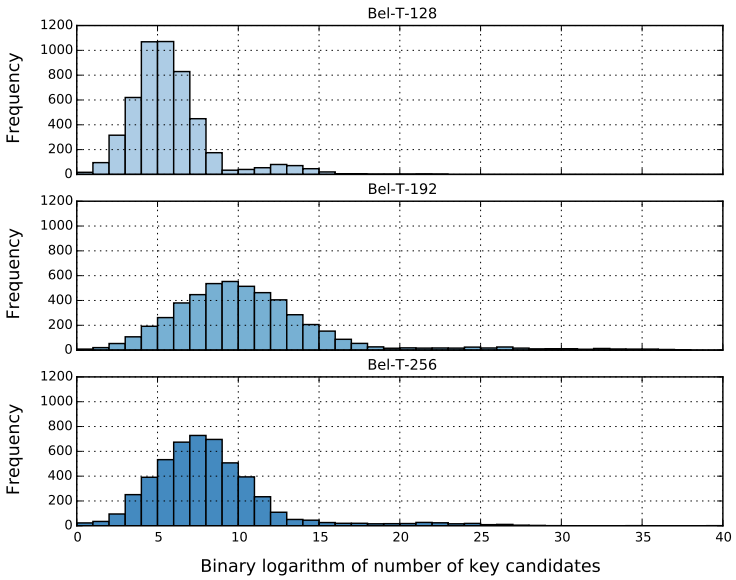


Figure 31: Experimental results for the fault analysis of Bel-T.

Some of the known countermeasures are effective against the above attack. Design obfuscation will at least complicate the task to identify the location and the time of fault injection. Light sensors aiming at detection of attempts to open the package will be effective against laser attacks which require deprocessing but not against EM pulses which act through the package. Voltage drop or current sensors will detect the attacks if they are placed close enough to the vulnerable structures. Concurrent error-detection schemes with a sufficient fault coverage will report the injected faults. Frequent regeneration of the secret key will provide protection only if the complete attack (1 fault-free run and 4, 7, or 10 runs with injected faults) cannot be finished with the same key. All these countermeasures are associated with some cost and techniques to circumvent them are known.

3.5 Conclusion

We presented a detailed fault analysis of the Bel-T block cipher family by showing for each of its three variants, differing by key sizes of 128-, 192-, and 256-bit, how to successfully mount attacks requiring only 4, 7 and 10 faults, respectively. The attack against Bel-T-128 can be performed completely under a very weak fault model (RFM). The attacks on Bel-T-192 and Bel-T-256 using RFM faults recover only 96 and 128 bits of the 192- and

256-bit master key. Under a slightly stronger fault model (CFM) the security of the latter two variants collapses nevertheless and the entire secret key can be reconstructed. Our experiments show that all attacks yield compact sets of key candidates for which brute-force search is practical.

Chapter 4

NORX: Parallel and Scalable Authenticated Encryption

4.1 Introduction

This chapter introduces NORX a novel authenticated encryption scheme supporting associated data (AEAD). NORX has been submitted in early 2014 as a first round candidate to CAESAR, a cryptographic competition which aims to find the next generation of authenticated encryption primitives.

Cryptographic competitions [92] have a long-standing tradition in the cryptographic community. The very first was announced in 1997 by the National Institute of Standard and Technologies (NIST) of the United States of America, with the aim to replace the ageing *Data Encryption Standard (DES)* by a new *Advanced Encryption Standard (AES)*. A total of 15 designs were submitted to the first round of the competition and five of them advanced to the second, namely MARS [132], RC6 [210], Rijndael [95], Serpent [6], and Twofish [219]. Finally, in October 2000, NIST chose the algorithm Rijndael, designed by Daemen and Rjimen, two Belgian cryptographers, to become the AES [96] and thereby ending the contest.

Beyond the selection of the AES, this first cryptographic competition is generally viewed as a huge success for the entire field of cryptology and in particular for secret-key cryptography. Especially its openness was a completely new approach and turned out to be very appropriate for the task of finding new cryptographic algorithms suitable for widespread adoption. The openness had major advantages over the common approach used until that time: prior to that, most cryptographic algorithms were created in secret, i.e. the design process was completely non-transparent, and afterwards specifications were kept proprietary. For example, after the Second World War, cryptographic algorithms were considered military equipment in the United States and it was illegal to sell or distribute encryption technology to foreign countries. All these points made it very hard to analyse cryptographic algorithms and assess their security. As a consequence, many of those designs experienced little or even no 3rd-party cryptanalysis at all, but still found their way into production environment, which in turn led to countless security

disasters. In comparison, algorithms submitted to the AES competition were designed by leading and well-known experts of the field and surveyed by the cryptographic community over many years. The transparency of the design process and the continuous analysis of the algorithms obviously led to a much better understanding of their strengths and weaknesses. It also generated a very high level of trust in the security of the five finalists, which remain (practically) unbroken to the present day.

Due to the success of the format, it is not very surprising that new competitions emerged shortly thereafter: in 2004, *the ECRYPT Stream Cipher Project (eSTREAM)* [107] searched for new and innovative stream cipher designs for hard- and software architectures. After its announcement, 34 designs were submitted to the first round and eight made it to the final selection, which were divided in two portfolios: HC-128 [245], Rabbit [69], Salsa20 [41], and SOSEMANUK [34] for software and F-FCSR [10], Grain [2], MICKEY [22], and Trivium [86] for hardware. However, F-FCSR was broken shortly after the announcement of the final selection and was therefore removed from the latter.

The next contest, the *SHA-3 Competition* [222], was announced by NIST in 2007 and ended in 2013. The main motivation was a series of attacks in 2004 and 2005 on the hash functions MD5 and SHA-0 that found practical collisions [57, 239, 241]. Yet another result even showed how to construct collisions in SHA-1 [240] requiring approximately 2^{69} operations. Although the above attack was only a theoretical break, it became clear that it would be only a matter of time until a first practical collision would have been found for SHA-1, too. Another concern was the proximity of the SHA-2 design to MD5, SHA-0, and SHA-1, which posed the threat that attacks on the latter three algorithms are also expandable to SHA-2. Thus, a demand for action was obviously necessary, in order to have a good fall-back algorithm in case of another unexpected cryptanalytic advancement on the SHA hash function family, which then might even threaten the security of SHA-2. A total of 51 designs were accepted to the first round and five made it to the finals: BLAKE [16], Grøstl [118], JH [246], Keccak [47], and Skein [112]. Finally, in 2013, NIST selected Keccak as the winner of the contest, due to its very high speed in hardware and its structural differences from SHA-2. At the time of writing this thesis, Keccak is under standardisation to become the new SHA-3 standard.

The latest two iterations of contests, announced in 2013, feature the *Competition for Authenticated Encryption: Security, Applicability, and Robustness (CAESAR)* [84] and the *Password Hashing Competition (PHC)* [206]. Both of them officially started in 2014.

The motivation behind the PHC is the poor state-of-the-art of password hashing schemes and the low variety of available methods. Just a single scheme was standardised by NIST, namely PBKDF2 [147], and beyond that only two more alternatives exist: bcrypt [209] and scrypt [204]. Normal cryptographic hash functions, like the (still) widely used MD-5 and SHA-1, are completely unsuitable for the task of password hashing, since they have no protection against dedicated brute-force cracking methods running on graphics processing units (GPU), field-programmable gate arrays (FPGA) or application-specific integrated circuits (ASIC). An attacker who has access to specialised hardware

implementing such hash functions can enumerate exhaustively a huge search space very efficiently by exploiting massive parallelism and thus has a large advantage over the defender who deems the password protected through the hash. However, any good password hashing scheme should neutralise the advantage of an attacker who is trying to brute-force a password through the employment of specialised hardware. PBKDF2, bcrypt and scrypt all provide this feature in one way or the other, but there is still lots of room for improvements. We do not go into further details at this point and refer the interested reader to the official PHC website [206] instead.

CAESAR, the second competition that started in 2014, invited cryptographers to submit authenticated encryption schemes supporting associated data that offer advantages over AES-GCM [96, 196], the current de-facto standard, and are suitable for widespread adoption. An authenticated encryption (AE) scheme provides not only confidentiality of the processed data but also ensures its integrity and authenticity. An adversary intercepting a message protected in such a way can neither learn something about the plaintext data nor can he modify the transmitted data without being detected. Finally, the receiver can also verify that the data originates from a legitimate source using the shared secret keys. An extension of AE are authenticated encryption schemes with associated data (AEAD) [211]. Associated data (AD) is only authenticated during processing but not encrypted, i.e. authenticity and integrity of the associated data are ensured but it is transmitted in clear. Associated data can take many forms like meta data or routing information in TCP/IP packets. A more detailed introduction to the topic of authentication encryption is given in Section 1.1.5.

Over time serious mistakes were committed in many cases during design, implementation, and application of AE(AD) schemes, which led to numerous security disasters. To name just a few, as listed on the cryptographic competitions website [92]: in 2007, an attack on the *Wired Equivalent Privacy* (WEP) standard [232], as used in many 802.11 Wi-Fi networks, was presented. The attack exploits weaknesses in RC4, the underlying cipher, and allows to recover the secret WEP key from a few intercepted ciphertext messages within minutes. In 2009, an attack on the OpenSSH protocol [4] showed how to recover 32 plaintext bits with probability 2^{-18} at a position freely selectable by the attacker. The problem that made this attack possible was a flaw in the interaction between encryption and authentication mechanisms as used by OpenSSH. In 2012, a flaw was found in EAXprime [185], an authenticated encryption block cipher mode standardised by ANSI under the number C12.22-2008 for smart grid applications that was even subject of a forthcoming NIST standard. EAXprime can be instantiated with an arbitrary block cipher, like AES, and transforms the latter into an authenticated encryption mode. The flaw can be exploited to mount a fast forgery attack on EAXprime, which allows an adversary to create ciphertexts that seem valid even though they were not created by the actual encryptor. Moreover, the attack works independently of the underlying block cipher. In 2015, researchers disclosed severe vulnerabilities [142, 174] in the cryptographic infrastructure of the Open Smart Grid Protocol (OSGP) which uses an AE scheme based

on RC4 and a self-made MAC. The differential attacks discussed in [142] focus on the MAC and are particularly critical. Some of them fall into the ciphertext-only attack category and allow to reconstruct the secret key used in OSGP's smart-meters in a practical scenario.

Due to the many problems with authenticated encryption schemes, the cryptographic community decided that new solutions need to be found, which are safer, faster, and easier to use than the existing options and especially AES-GCM. These ambitions culminated in the mentioned CAESAR competition.

The first round of CAESAR accepted 57 submissions and the chapter at hand presents NORX[1], our candidate for the competition. NORX is a state-of-the-art authenticated encryption scheme supporting associated data. It relies on well-known building blocks proven successful through years of cryptanalysis and which have been improved further during the design phase to meet our objectives. The layout of NORX is based on the *monkeyDuplex* construction [44, 48], extended by the capability to process data in parallel and an (almost) freely tunable parallelism degree. The monkeyDuplex construction belongs to the family of so-called *Sponge functions*, developed alongside of Keccak [47], the winner of the SHA-3 competition. Thanks to the duplex construction, the size of the authentication tag can be adapted very easily to meet the requirements of the application at hand. An original domain separation scheme allows simple processing of header, payload and trailer data. NORX was optimized for efficiency in both soft- and hardware, with a SIMD-friendly core, almost byte-aligned rotations and no secret-dependent memory lookups.

The NORX core traces its legacy back to the ARX primitives Salsa20 [41], ChaCha [40], and BLAKE(2) [16, 21]. Salsa20 is a stream cipher designed by Bernstein in 2005 and is a member of the final portfolio of the eSTREAM contest mentioned above. ChaCha, the successor of Salsa20, is another step forward in terms of security and speed and is capable to even exceed the performance of AES implementations using the *Advanced Encryption Standard New Instructions* (AES-NI) by Intel [122]. BLAKE and BLAKE2 are ChaCha-based hash functions, designed by Aumasson et al. BLAKE is one of the five finalists of the SHA-3 contest and known for its high security margin and very good speed in software. BLAKE2 improves even more on the software performance, aiming to provide a secure and faster alternative to MD5, which is still widely used due to its high software speed despite beeing insecure. The NORX core function is very close to that of ChaCha et al. but replaces integer addition with an approximation of the latter exclusively based on bitwise logical operations. The intention behind this design decision was to simplify security analysis, improve hardware efficiency and provide resistance to timing attacks. Furthermore, NORX specifies a dedicated datagram to facilitate interoperability, protect the users from the trouble of defining custom encoding and signalling, and to simplify integration into existing protocol stacks. Measurements show that the performance of

[1] The name stems from "NO(T A)RX" and is pronounced like "norcks".

NORX is very good in both soft- and hardware. For example, the results from the SUPERCOP [230] software benchmarking suite for cryptographic primitives show that NORX is very fast on a broad range of platforms. In fact, it is among the top algorithms of all the CAESAR submissions and it seems to be the fastest Sponge-based scheme on most of the examined platforms.

The NORX family of authenticated encryption schemes was presented at ESORICS 2014 [20]. Additionally, we gave a talk on CAESAR and NORX at the 31st Chaos Communication Congress [17].

Outline. Section 4.2 gives a complete specification of the NORX family of AEAD schemes. Section 4.3 presents security goals and discusses results on the security bounds of the NORX mode of operation. In Section 4.4, we first discuss the features of NORX, then we justify the choice of the cipher's parameters and finally report on performance measurements: we show results of our software evaluation on 32- and 64-bit processors and describe the results of an evaluation of the scheme on an ASIC performed by an external party. Section 4.5 motivates design decisions. Finally, in Section 4.6, we conclude the chapter and give an outlook on possible future developments.

4.2 Specification

4.2.1 Preliminaries

In the following, we introduce techniques to convert bit strings to integer vectors and vice versa which are used in the specification of NORX.

Let $m, n \in \mathbb{N}$ and let $v = (v_0, \ldots, v_{m-1}) \in \mathbb{Z}_{2^l}^m$ and $w = (w_0, \ldots, w_{n-1}) \in \mathbb{Z}_{2^l}^n$ be l-bit *integer vectors* of length m and n, respectively. We denote the concatenation of v and w by $v \parallel w = (v_0, \ldots, v_{m-1}, w_0, \ldots, w_{n-1}) \in \mathbb{Z}_{2^l}^{m+n}$. Now let $l, m, n \in \mathbb{N}$ such that $n = lm$. Then we define the function

$$\mathsf{to_int}_l : \mathbb{F}_2^n \to \mathbb{Z}_{2^l}^m,\ x_0 \parallel \cdots \parallel x_{n-1} \mapsto (v_0, \ldots, v_{m-1})$$

with

$$v_i = \sum_{j=0}^{l-1} 2^j x_{il+j}$$

for all $i \in \{0, \ldots, m-1\}$ which converts an n-bit string to a vector of l-bit integers of length m. For the other direction, we define

$$\mathsf{to_str}_l : \mathbb{Z}_{2^l}^m \to \mathbb{F}_2^n,\ (v_0, \ldots, v_{m-1}) \mapsto x_0 \parallel \cdots \parallel x_{m-1}$$

with

$$x_i = \frac{v_i}{2^0} \bmod 2 \parallel \cdots \parallel \frac{v_i}{2^{l-1}} \bmod 2$$

for all $i \in \{0, \ldots, m-1\}$ which converts a vector of l-bit integers of length m to a bit string of length n. It obviously holds that $\mathsf{to_str}_l(\mathsf{to_int}_l(X)) = X$ for an arbitrary n-bit string X with $n = lm$.

4.2.2 Parameters and Interface

The NORX family of authenticated encryption schemes is parametrised by a *wordsize* w of 32 or 64 bits, a *number of rounds* $1 \leq \mathsf{r} \leq 63$, a *parallelism degree* $0 \leq \mathsf{d} \leq 255$, and a *tag size* of $\mathsf{t} \leq 10\mathsf{w}$ bits, with a default of $\mathsf{t} = 4\mathsf{w}$ bits.

Encryption Mode

A NORX instance in encryption mode takes as input, a *secret key* $K \in \mathbb{F}_2^{\mathsf{k}}$ with $\mathsf{k} = 4\mathsf{w}$, a *nonce* $N \in \mathbb{F}_2^{\mathsf{n}}$ with $\mathsf{n} = 2\mathsf{w}$, and a *message* $M = A \parallel P \parallel B$, where $A \in \mathbb{F}_2^*$ is a *header*, $P \in \mathbb{F}_2^*$ is a *payload*, and $B \in \mathbb{F}_2^*$ is a *trailer*. A and B are both considered as *associated data*. NORX encryption produces a *ciphertext* (or *encrypted payload*) $C \in \mathbb{F}_2^*$ of the same size as P and an *authentication tag* $T \in \mathbb{F}_2^{\mathsf{t}}$. In summary, NORX encryption $\mathcal{E}$ is specified as

$$\mathcal{E} : \mathbb{F}_2^{\mathsf{k}} \times \mathbb{F}_2^{\mathsf{n}} \times \mathbb{F}_2^* \times \mathbb{F}_2^* \times \mathbb{F}_2^* \to \mathbb{F}_2^* \times \mathbb{F}_2^{\mathsf{t}}$$

with

$$\mathcal{E}_K(N, A, P, B) = (C, T)$$

where $|P| = |C|$.

Decryption Mode

A NORX instance in decryption mode takes as input a secret key $K \in \mathbb{F}_2^{\mathsf{k}}$ with $\mathsf{k} = 4\mathsf{w}$ bits, a nonce $N \in \mathbb{F}_2^{\mathsf{n}}$ with $\mathsf{n} = 2\mathsf{w}$ bits, a message $M = A \parallel C \parallel B$, where $A \in \mathbb{F}_2^*$ is a header, $C \in \mathbb{F}_2^*$ is an encrypted payload, and $B \in \mathbb{F}_2^*$ is a trailer, and an authentication tag $T \in \mathbb{F}_2^{\mathsf{t}}$. NORX decryption either returns an error $\perp$, upon failed verification of the authentication tag, or produces a plaintext P of the same size as C if tag verification succeeds. In summary, NORX decryption $\mathcal{D}$ is specified by

$$\mathcal{D} : \mathbb{F}_2^{\mathsf{k}} \times \mathbb{F}_2^{\mathsf{n}} \times \mathbb{F}_2^* \times \mathbb{F}_2^* \times \mathbb{F}_2^* \times \mathbb{F}_2^{\mathsf{t}} \to \mathbb{F}_2^* \cup \{\perp\}$$

with

$$\mathcal{D}_K(N, A, C, B, T) = \begin{cases} P & \text{if } T = T' \\ \perp & \text{if } T \neq T' \end{cases}$$

where T denotes the received authentication tag, T' the one computed on the recipient's side and $|P| = |C|$.

Naming Conventions

A NORX instance is denoted by NORXw-r-d-t , where w, r, d, and t are the parameters of the instance as introduced above. If the default tag size is used, i.e. $\mathsf{t} = 4\mathsf{w}$, the notation for an instance is shortened to NORXw-r-d . So for example, NORX64-6-1 has $(\mathsf{w}, \mathsf{r}, \mathsf{d}, \mathsf{t}) = (64, 6, 1, 256)$.

Instances

We propose five concrete instances of NORX, which are specified in Table 14.

Table 14: NORX instances.

w	r	d	t	k	n
64	4	1	256	256	128
32	4	1	128	128	64
64	6	1	256	256	128
32	6	1	128	128	64
64	4	4	256	256	128

All instances use the default tag size of 4w bits, i.e. 128 bit for NORX32 and 256 bit for NORX64. Table 14 also reflects the priority order of the recommended parameter sets from highest on the top (NORX64-4-1) to lowest at the bottom (NORX64-4-4). A more detailed discussion on those parameter combinations can be found in Section 4.4.2.

4.2.3 Layout Overview

NORX relies on the monkeyDuplex construction [44, 48], enhanced with the capability of parallel payload processing. The number l of parallel *encryption lanes* L_i, with $0 \leq i \leq l-1$, is controlled by the parameter $0 \leq \mathsf{d} \leq 255$. For the value $\mathsf{d} = 1$, the layout of NORX corresponds to a standard (sequential) duplex construction, see Figure 32.

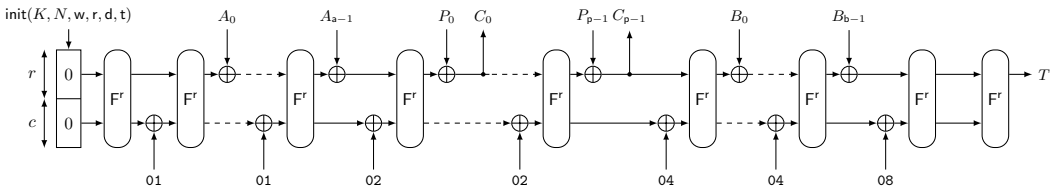


Figure 32: Layout of NORX for $\mathsf{d} = 1$.

For $\mathsf{d} > 1$, the number of lanes l is bounded by the latter value, e.g. for $\mathsf{d} = 2$ see

Figure 33. If $\mathsf{d} = 0$, the number of lanes l is bounded by the size of the payload. In that case, the layout of NORX is similar to the PPAE construction [63].

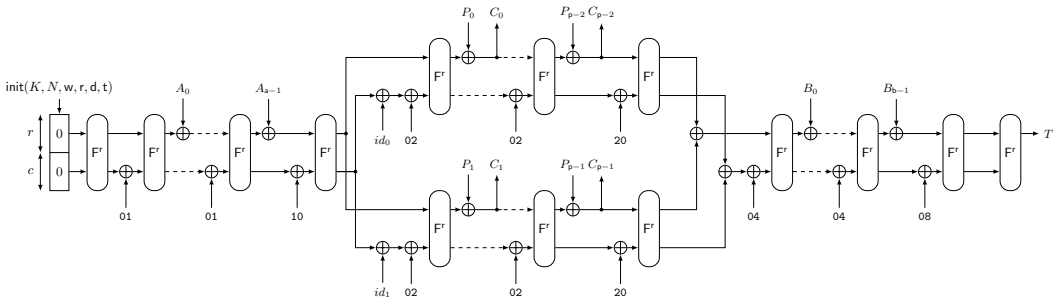


Figure 33: Layout of NORX for $\mathsf{d} = 2$.

The core algorithm F of NORX is a permutation of $b = r + c$ bits, where b is called the *width*, r the *rate* (or *block length*), and c the *capacity*. We call F the *round function* and F^{r} denotes its r-fold iteration. The internal state s of NORX64 has $b = 640 + 384 = 1024$ bits and that of NORX32 has $b = 320 + 192 = 512$ bits. The state is viewed as vector of 16 w-bit sized words, i.e. $s = (s_0, \ldots, s_{15}) \in \mathbb{Z}_{2^{\mathsf{w}}}^{16}$, and is conceptually arranged in a 4×4 matrix as shown below

$$s = \left(\begin{array}{cccc} s_0 & s_1 & s_2 & s_3 \\ s_4 & s_5 & s_6 & s_7 \\ s_8 & s_9 & s_{10} & s_{11} \\ \hline s_{12} & s_{13} & s_{14} & s_{15} \end{array}\right)$$

where $s_0, \ldots, s_9$ are called *rate words*, used for absorbing and extracting of data, and $s_{10}, \ldots, s_{15}$ are called *capacity words*, which remain untouched during data absorption or extraction.

4.2.4 The Round Function

The NORX round F, see Algorithm 6, processes a state $s \in \mathbb{Z}_{2^{\mathsf{w}}}^{16}$ by first transforming its columns and then transforming its diagonals with a permutation G, see Algorithm 7. Those two operations are called *column step* and *diagonal step*, as in BLAKE2 [21], and we denote them by col and diag, respectively. An illustration of the function F is shown in Figure 34.

The rotation offsets (r_0, r_1, r_2, r_3) as used in the 32- and 64-bit G function of NORX are specified in Table 15 and were chosen as discussed in Section 4.5.3. A visualisation of the G circuit is given in Figure 35. An important component of G is the non-linear

Algorithm 6: $\mathsf{F}(s)$

Inputs:
$s \in \mathbb{Z}_{2^\mathsf{w}}^{16}$
Outputs:
$s \in \mathbb{Z}_{2^\mathsf{w}}^{16}$
Algorithm:
1. $s_0,\ s_4,\ s_8, s_{12} \leftarrow \mathsf{G}(s_0, s_4, s_8, s_{12})$
2. $s_1,\ s_5,\ s_9, s_{13} \leftarrow \mathsf{G}(s_1, s_5, s_9, s_{13})$
3. $s_2, s_6, s_{10}, s_{14} \leftarrow \mathsf{G}(s_2, s_6, s_{10}, s_{14})$
4. $s_3, s_7, s_{11}, s_{15} \leftarrow \mathsf{G}(s_3, s_7, s_{11}, s_{15})$
5. $s_0, s_5, s_{10}, s_{15} \leftarrow \mathsf{G}(s_0, s_5, s_{10}, s_{15})$
6. $s_1, s_6, s_{11}, s_{12} \leftarrow \mathsf{G}(s_1, s_6, s_{11}, s_{12})$
7. $s_2,\ s_7,\ s_8, s_{13} \leftarrow \mathsf{G}(s_2, s_7, s_8, s_{13})$
8. $s_3,\ s_4,\ s_9, s_{14} \leftarrow \mathsf{G}(s_3, s_4, s_9, s_{14})$
9. **return** s

Algorithm 7: $\mathsf{G}(a, b, c, d)$

Inputs:
$a, b, c, d \in \mathbb{Z}_{2^\mathsf{w}}$
Outputs:
$a, b, c, d \in \mathbb{Z}_{2^\mathsf{w}}$
Algorithm:
1. $a \leftarrow (a \oplus b) \oplus ((a \wedge b) \ll 1)$
2. $d \leftarrow (a \oplus d) \ggg r_0$
3. $c \leftarrow (c \oplus d) \oplus ((c \wedge d) \ll 1)$
4. $b \leftarrow (b \oplus c) \ggg r_1$
5. $a \leftarrow (a \oplus b) \oplus ((a \wedge b) \ll 1)$
6. $d \leftarrow (a \oplus d) \ggg r_2$
7. $c \leftarrow (c \oplus d) \oplus ((c \wedge d) \ll 1)$
8. $b \leftarrow (b \oplus c) \ggg r_3$
9. **return** a, b, c, d

operation H (see Algorithm 7), which is specified as follows:

$$\mathsf{H} : \mathbb{F}_2^{2\mathsf{w}} \to \mathbb{F}_2^{\mathsf{w}}, (x, y) \mapsto (x \oplus y) \oplus ((x \wedge y) \ll 1) \ .$$

A thorough discussion on the design of the functions F, G, and H is given in Section 4.5.2.

Table 15: Rotation offsets for 32- and 64-bit NORX.

w	r_0	r_1	r_2	r_3
32	8	11	16	31
64	8	19	40	63

4.2.5 Encryption Mode

NORX encryption can process messages M of the form $M = A \parallel P \parallel B$, where A denotes a *header*, P a *payload* and B a *trailer*. A and B are also called *associated data*. Each of A, P and B are allowed to be empty strings of bits.

Structure

NORX encryption and authentication is depicted in Algorithm 8 and consists of multiple processing phases. For a more visual representation see Figures 32 and 33. After *initialisation*, where basically the state is loaded with the key and the nonce, processing of a message $M = A \parallel P \parallel B$ is done in up to five steps, namely *header processing*, *branching*, *payload encryption*, *merging*, and *trailer processing*. The number of steps depends on whether A, B, or P are empty or not and whether $\mathsf{d} = 1$ or not. NORX skips

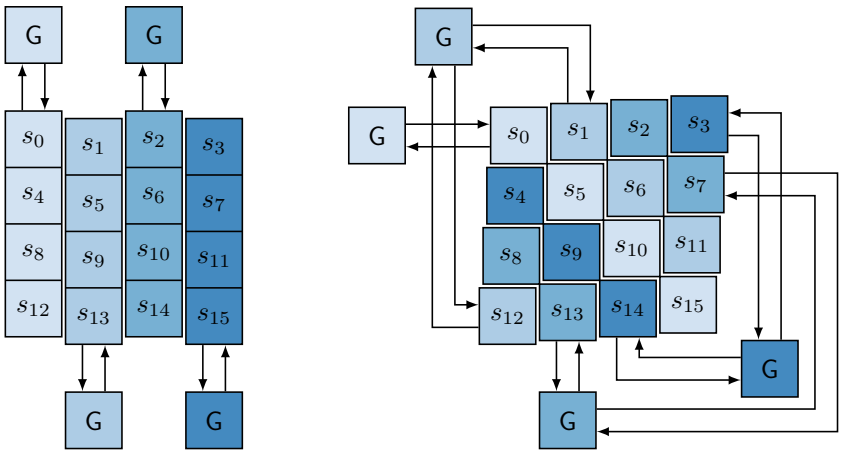


Figure 34: Column step and diagonal step of F.

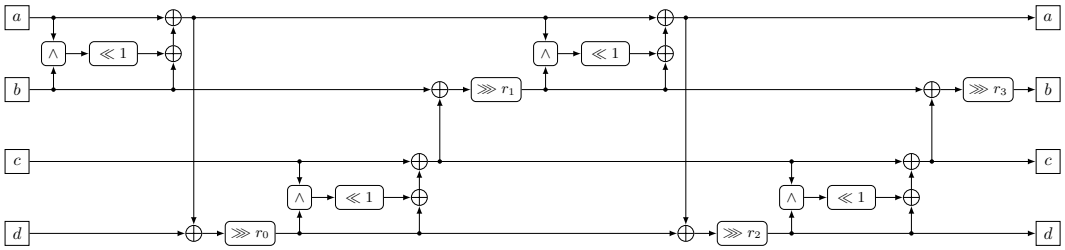


Figure 35: The G circuit.

processing phases of empty message parts and, if the cipher is parametrised with $\mathsf{d} = 1$, also branching and merging phases. For example, in the simple case when $|A| = |B| = 0$, $|P| > 0$, and $\mathsf{d} = 1$, message processing is done in one step since only the payload P needs to be encrypted and authenticated. Finally, after M has been processed, the *tag generation* phase produces the authentication tag which ensures integrity of the ciphertext C and of associated data A and B, and also allows to check authenticity of the data on the receiver's side.

Below, we first describe the padding and domain separation rules and afterwards each of the aforementioned phases.

Padding

NORX adopts the so-called *multi-rate padding*, which is a *sponge compliant padding* as introduced in [45, 48]. This padding rule is defined by the map

$$\mathsf{pad}_r : \mathbb{F}_2^n \to \mathbb{F}_2^{n+q+2}, X \longmapsto X \parallel 10^q 1_2$$

Algorithm 8: norx_encrypt(K, N, A, P, B)

Inputs:
$K \in \mathbb{F}_2^{4w}$, $N \in \mathbb{F}_2^{2w}$, $A \in \mathbb{F}_2^*$, $P \in \mathbb{F}_2^*$, $B \in \mathbb{F}_2^*$
Outputs:
$C \in \mathbb{F}_2^*$, with $|P| = |C|$, $T \in \mathbb{F}_2^t$
Algorithm:
1. $s \leftarrow \mathsf{initialise}(K, N)$
2. $s \leftarrow \mathsf{absorb_data}(s, A, 01)$
3. $s \leftarrow \mathsf{branch}(s, |\mathsf{pad}_r(P)|)$
4. $s, C \leftarrow \mathsf{encrypt_data}(s, P, 02)$
5. $s \leftarrow \mathsf{merge}(s)$
6. $s \leftarrow \mathsf{absorb_data}(s, B, 04)$
7. $s, T \leftarrow \mathsf{generate_tag}(s)$
8. **return** C, T

with bit strings X and 10^q1_2, and $q = -(|X| + 2) \bmod r$. The multi-rate padding extends X to a multiple of the rate r, never results in the empty string, and guarantees that the last block of $\mathsf{pad}_r(X)$ differs from the all-zero block 0^r. There are three special cases:

$$q = \begin{cases} r-2, & \text{if} \quad 0 \equiv |X| \bmod r \\ 0, & \text{if} \quad r-2 \equiv |X| \bmod r \\ r-1, & \text{if} \quad r-1 \equiv |X| \bmod r \ . \end{cases}$$

In the first case, $|X|$ is a multiple of the rate r and thus a full r-bit sized block $10^{(r-2)}1_2$ is appended to X. This includes the case where X is the empty bit string ε, i.e. if $|X| = 0$. The second and third cases describe the situations where the smallest and largest numbers of bits are appended to X, respectively. This corresponds to the padding block 11_2 of size 2 in the former and the padding block $10^{(r-1)}1_2$ of size $r + 1$ in the latter case.

Domain Separation

NORX has a very simple and lightweight domain separation mechanism: it is performed by XORing a *domain separation constant* to the least significant byte of s_{15} each time before the state s is transformed by the permutation F^r. Distinct constants are used for the different algorithm phases, i.e. for the three different message processing stages, for tag generation, and in case of $\mathsf{d} \neq 1$, for branching and merging steps. Table 16 gives the specification of those constants and Figures 32 and 33 illustrate their integration into the state of NORX.

The type of the domain separation constant used at a particular step is determined by the type of the *next* processed data block. The constants are switched together with the phases. For example, as long as the next block is from the header, the domain separation constant 01 is applied. Once all header blocks are processed, the constant is switched. If

Table 16: Domain separation constants.

header	payload	trailer	tag	branching	merging
01	02	04	08	10	20

$\mathsf{d} = 1$ and the next data block belongs to the payload, the new constant is 02. Then, as long as the next block is from the payload, 02 is used and so on.

This technique also allows NORX to skip unneeded processing phases. For example, if $\mathsf{d} = 1$, $|A| > 0$, $|P| > 0$, and $|B| = 0$, the constant 08 is integrated during processing of the *last* payload block, which means that the trailer phase is skipped and NORX advances directly to the generation of the authentication tag. For the extra initial and final permutations no domain separation constants are used, which is equivalent to XORing 00 to s_{15}. For the special case $\mathsf{d} \neq 1$ and $|P| = 0$ not only payload processing is skipped but also branching and merging phases.

Initialisation

NORX initialisation processes a 4w-bit key K, a 2w-bit nonce N, and the instance parameters d, r, w, and t. First, the state $s = (s_0, \ldots, s_{15}) \in \mathbb{Z}_{2^\mathsf{w}}^{16}$ is initialised, followed by loading key $k = (k_0, k_1, k_2, k_3) = \mathsf{to_int}_\mathsf{w}(K)$, nonce $n = (n_0, n_1) = \mathsf{to_int}_\mathsf{w}(N)$, and constants $u = (u_0, \ldots, u_9)$ into s. The values of $u_0, \ldots, u_3$ for NORX32 (left) and NORX64 (right) are specified as

$$\begin{aligned} u_0 &= \texttt{243f6a88} \\ u_1 &= \texttt{85a308d3} \\ u_2 &= \texttt{13198a2e} \\ u_3 &= \texttt{03707344} \end{aligned} \qquad \begin{aligned} u_0 &= \texttt{243f6a8885a308d3} \\ u_1 &= \texttt{13198a2e03707344} \\ u_2 &= \texttt{a4093822299f31d0} \\ u_3 &= \texttt{082efa98ec4e6c89} \end{aligned}$$

and the remaining values are computed by

$$(u_{4j+4}, u_{4j+5}, u_{4j+6}, u_{4j+7}) = \mathsf{G}(u_{4j}, u_{4j+1}, u_{4j+2}, u_{4j+3})$$

for $j \in \{0, 1\}$. A complete list of the constants is given in Table 23. Afterwards, the parameters d, r, w, and t are integrated into s by XORing them to the words s_{12}, s_{13}, s_{14}, and s_{15}, respectively. Finally, the state s is updated with r iterations of the round function F. Algorithm 9 shows the steps executed during the initialisation phase of NORX.

Message Processing

Message processing is the main phase of NORX encryption or decryption. Unless noted otherwise, the value of the domain separation constant is always determined as described above.

Algorithm 9: $\mathsf{initialise}(K, N)$

Inputs:
$K \in \mathbb{F}_2^{4\mathsf{w}}$, $N \in \mathbb{F}_2^{2\mathsf{w}}$
Outputs:
$s \in \mathbb{Z}_{2^\mathsf{w}}^{16}$
Algorithm:
1. $s \leftarrow \mathsf{to_int}_\mathsf{w}(0^{16\mathsf{w}})$
2. $k \leftarrow \mathsf{to_int}_\mathsf{w}(K)$
3. $n \leftarrow \mathsf{to_int}_\mathsf{w}(N)$
4. $s_0, \ s_1, \ s_2, \ s_3 \leftarrow u_0, n_0, n_1, u_1$
5. $s_4, \ s_5, \ s_6, \ s_7 \leftarrow k_0, k_1, k_2, k_3$
6. $s_8, \ s_9, s_{10}, s_{11} \leftarrow u_2, u_3, u_4, u_5$
7. $s_{12}, s_{13}, s_{14}, s_{15} \leftarrow u_6, u_7, u_8, u_9$
8. $s_{12} \leftarrow s_{12} \oplus \mathsf{w}$
9. $s_{13} \leftarrow s_{13} \oplus \mathsf{r}$
10. $s_{14} \leftarrow s_{14} \oplus \mathsf{d}$
11. $s_{15} \leftarrow s_{15} \oplus \mathsf{t}$
12. $s \leftarrow \mathsf{F}^\mathsf{r}(s)$
13. **return** s

Header Processing. If $|A| = 0$, this stage is skipped, otherwise A is padded to a multiple of r bits using the multi-rate padding, i.e. $A' = \mathsf{pad}_r(A)$. Let A'_i denote the ith r-bit sized header block of A' with $0 \leq i \leq |A'|_r - 1$. Before A'_i is processed, the domain separation constant `01` is added to s_{15} using bitwise `XOR` followed by an update of s by F^r. Then, A'_i is converted to a vector of integers using $\mathsf{to_int}_\mathsf{w}$ and `XOR`'ed to the rate words of s. The two functions absorb_block and absorb_data required for header processing are described in detail in Algorithms 10 and 11.

Algorithm 10: $\mathsf{absorb_data}(s, X, v)$

Inputs:
$s \in \mathbb{Z}_{2^\mathsf{w}}^{16}$, $X \in \mathbb{F}_2^*$, $v \in \mathbb{Z}_{2^8}$
Outputs:
$s \in \mathbb{Z}_{2^\mathsf{w}}^{16}$
Algorithm:
1. **if** $|X| > 0$ **then**
2. $\quad X' \leftarrow \mathsf{pad}_r(X)$
3. $\quad$ **for** $i \in \{0, \ldots, |X'|_r - 1\}$ **do**
4. $\quad\quad s \leftarrow \mathsf{absorb_block}(s, X'_i, v)$
5. $\quad$ **end**
6. **end**
7. **return** s

Algorithm 11: $\mathsf{absorb_block}(s, X, v)$

Inputs:
$s \in \mathbb{Z}_{2^\mathsf{w}}^{16}$, $X \in \mathbb{F}_2^r$, $v \in \mathbb{Z}_{2^8}$
Outputs:
$s \in \mathbb{Z}_{2^\mathsf{w}}^{16}$
Algorithm:
1. $x \leftarrow \mathsf{to_int}_\mathsf{w}(X)$
2. $s_{15} \leftarrow s_{15} \oplus v$
3. $s \leftarrow \mathsf{F}^\mathsf{r}(s)$
4. **for** $i \in \{0, \ldots, r/\mathsf{w} - 1\}$ **do**
5. $\quad s_i \leftarrow s_i \oplus x_i$
6. **end**
7. **return** s

Branching. This phase is omitted if NORX is parametrised with $\mathsf{d} = 1$. Otherwise, if $\mathsf{d} \neq 1$, the state s is prepared for parallel payload encryption. First, the domain separation

constant 10 is included in s_{15}, followed by an update of s with F^r. Then, either $l = \mathsf{d}$ if $\mathsf{d} > 1$ or $l = |\mathsf{pad}_r(P)|_r$ if $\mathsf{d} = 0$, copies of s are created and copied to a sufficiently large vector s'. Finally, the lane id $i \in \{0, \ldots, l-1\}$ is integrated into words s'_{16i+13} (and possibly s'_{16i+14} for very large l) of each copy and s' is returned. Algorithm 12 summarises the steps executed during the branch operation.

Algorithm 12: $\mathsf{branch}(s, m)$

Inputs:
$s \in \mathbb{Z}_{2^w}^{16}$, $m \in \mathbb{N}$, with $m \geq r$
Outputs:
$s' \in \mathbb{Z}_{2^w}^{16l}$, with $l \geq 1$
Algorithm:
1. $l \leftarrow \mathsf{d}$
2. **if** $\mathsf{d} = 0$ **then**
3. $\quad l \leftarrow m/r$
4. **end**
5. $s' \leftarrow \mathsf{to_int}_w(0^{16\,l\mathsf{w}})$
6. **if** $\mathsf{d} \neq 1$ **then**
7. $\quad s_{15} \leftarrow s_{15} \oplus 10$
8. $\quad s \leftarrow \mathsf{F}^r(s)$
9. $\quad$ **for** $i \in \{0, \ldots, l-1\}$ **do**
10. $\quad\quad (s'_{16i}, \ldots, s'_{16i+15}) \leftarrow s$
11. $\quad\quad s'_{16i+13} \leftarrow s'_{16i+13} \oplus i \bmod 2^w$
12. $\quad\quad s'_{16i+14} \leftarrow s'_{16i+14} \oplus \lfloor i/2^w \rfloor$
13. $\quad$ **end**
14. **else**
15. $\quad s' \leftarrow s$
16. **end**
17. **return** s'

Payload Encryption. If $|P| = 0$, this stage is skipped. Otherwise, payload data is padded using the multi-rate padding and then encrypted. Let $\mathsf{pad}_r(P) = P'$ and let P'_i be the ith r-bit sized block of P' for $0 \leq i \leq |P'|_r - 1$. We distinguish three cases how the blocks P'_i are processed depending on the value of d:

- $\mathsf{d} = 1$: This is the standard case, which requires no special handling.
- $\mathsf{d} > 1$: In this case, a fixed number of lanes L_j is available for payload encryption, with $0 \leq j \leq \mathsf{d} - 1$. An r-bit sized block P'_i is processed by lane L_j if $j \equiv i \bmod \mathsf{d}$. In other words, the padded payload blocks are distributed through the lanes in a round-robin fashion.
- $\mathsf{d} = 0$: Here, the number of lanes L_i is determined by the number $|P'|_r$ of padded payload blocks. Each r-bit sized block is processed on its own lane, i.e. block P'_i is encrypted on L_i, with $0 \leq i \leq |P'|_r - 1$.

The data encryption of a single block works equivalently for each value of d, hence we describe it only in a generic way. As above, let $P' = \mathsf{pad}_r(P)$ be the padded payload. Before a block P'_i is processed, the domain separation constant $\mathtt{02}$ is integrated into s_{15} followed by an update of the state s with F^r. Afterwards, P'_i is converted to a vector of integers using $\mathsf{to_int}_\mathsf{w}$ and the resulting words are added to the rate words of s using bitwise $\mathtt{XOR}$. The result of the latter operations then forms a new ciphertext block C_i after being converted back to a bit string using $\mathsf{to_str}_\mathsf{w}$. Note that for the last block of index $i = |P'|_r - 1$ only a truncated ciphertext block is created such that the final encrypted payload C has the same length as unpadded P. In other words, padding bits are never written to C.

The two operations $\mathsf{encrypt_data}$ and $\mathsf{encrypt_block}$ used for payload encryption are shown in Algorithms 13 and 14. Note that in $\mathsf{encrypt_data}$ the vector s has a length of $16\,l$ elements, with $l \geq 1$. The value of l is determined through the parameter d. This enlarged vector is used to model the parallel lanes in NORX encryption. In other words, lane L_i operates for data encryption on the subvector $(s_{16i}, \ldots, s_{16i+15})$ of length 16, with $0 \leq i \leq l - 1$.

Algorithm 13: $\mathsf{encrypt_data}(s, X, v)$

Inputs:
$s \in \mathbb{Z}_{2^\mathsf{w}}^{16l}$, with $l \geq 1$, $X \in \mathbb{F}_2^*$, $v \in \mathbb{Z}_{2^8}$
Outputs:
$s \in \mathbb{Z}_{2^\mathsf{w}}^{16l}$, $C \in \mathbb{F}_2^*$, with $|X| = |C|$
Algorithm:
1. $C \leftarrow \varepsilon$
2. **if** $|X| > 0$ **then**
3. $\quad X' \leftarrow \mathsf{pad}_r(X)$
4. $\quad m \leftarrow \mathsf{d}$
5. $\quad$ **if** $\mathsf{d} = 0$ **then**
6. $\quad\quad m \leftarrow |X'|_r$
7. $\quad$ **end**
8. $\quad$ **for** $i \in \{0, \ldots, |X'|_r - 1\}$ **do**
9. $\quad\quad j = i \bmod m$
10. $\quad\quad s' \leftarrow (s_{16j}, \ldots, s_{16j+15})$
11. $\quad\quad s', Y \leftarrow \mathsf{encrypt_block}(s', X'_i, v)$
12. $\quad\quad (s_{16j}, \ldots, s_{16j+15}) \leftarrow s'$
13. $\quad\quad C \leftarrow C \parallel Y$
14. $\quad$ **end**
15. $\quad C \leftarrow \lfloor C \rfloor_{|X|}$
16. **end**
17. **return** s, C

Algorithm 14: $\mathsf{encrypt_block}(s, X, v)$

Inputs:
$s \in \mathbb{Z}_{2^\mathsf{w}}^{16}$, $X \in \mathbb{F}_2^r$, $v \in \mathbb{Z}_{2^8}$
Outputs:
$s \in \mathbb{Z}_{2^\mathsf{w}}^{16}$, $Y \in \mathbb{F}_2^r$
Algorithm:
1. $x \leftarrow \mathsf{to_int}_\mathsf{w}(X)$
2. $y \leftarrow \mathsf{to_int}_\mathsf{w}(0^r)$
3. $s_{15} \leftarrow s_{15} \oplus v$
4. $s \leftarrow \mathsf{F}^\mathsf{r}(s)$
5. **for** $i \in \{0, \ldots, r/\mathsf{w} - 1\}$ **do**
6. $\quad s_i \leftarrow s_i \oplus x_i$
7. $\quad y_i \leftarrow s_i$
8. **end**
9. $Y \leftarrow \mathsf{to_str}_\mathsf{w}(y)$
10. **return** s, Y

Merging. In the case $\mathsf{d} \neq 1$, this phase folds back an enlarged state vector $s \in \mathbb{Z}_{2^\mathsf{w}}^{16l}$, with $l \geq 1$, to a vector $s' \in \mathbb{Z}_{2^\mathsf{w}}^{16}$.[2] For $\mathsf{d} = 1$ this stage is skipped altogether.

Recall that in the parallel case each subvector $(s_{16i}, \ldots, s_{16i+15})$ of length 16, with $0 \leq i \leq l-1$, corresponds to the state used by lane L_i for encryption. First, the domain separation constant $\mathtt{20}$ is included into the elements s_{16i+15} for all $0 \leq i \leq l-1$, followed by an update of the ith subvector with F^r. Finally, all the subvectors $(s_{16i}, \ldots, s_{16i+15})$ are summed up using bitwise $\mathtt{XOR}$ to form the final state vector s'. Algorithm 15 shows the steps executed during the merge operation.

Algorithm 15: $\mathsf{merge}(s)$

Inputs:
$s \in \mathbb{Z}_{2^\mathsf{w}}^{16l}$, with $l \geq 1$
Outputs:
$s' \in \mathbb{Z}_{2^\mathsf{w}}^{16}$
Algorithm:
1. $s' \leftarrow \mathsf{to_int_w}(0^{16\mathsf{w}})$
2. **if** $\mathsf{d} \neq 1$ **then**
3. **for** $i \in \{0, \ldots, l-1\}$ **do**
4. $x \leftarrow (s_{16i}, \ldots, s_{16i+15})$
5. $x_{15} \leftarrow x_{15} \oplus \mathtt{20}$
6. $x \leftarrow \mathsf{F}^\mathsf{r}(x)$
7. **for** $j \in \{0, \ldots, |x|-1\}$ **do**
8. $s'_j \leftarrow s'_j \oplus x_j$
9. **end**
10. **end**
11. **else**
12. $s' \leftarrow s$
13. **end**
14. **return** s'

Trailer Processing. Absorption of trailer data is done analogously to the processing of header data as already described above. Hence, if $|B| = 0$, trailer processing is skipped. If B is non-empty, let $B' = \mathsf{pad}_r(B)$ and let B'_i be the ith r-bit sized substring of B', with $0 \leq i \leq |B'|_r - 1$. Before a block is processed, the domain separation constant $\mathtt{04}$ is added to s_{15} using bitwise $\mathtt{XOR}$, followed by an update of s with F^r. Afterwards B'_i is converted to a vector of integers using $\mathsf{to_int_w}$ and absorbed into the rate part of s. The two required functions absorb_data and absorb_block are described in detail in Algorithms 10 and 11. After the trailer has been absorbed, the message processing part is also finished.

[2] Note that we need to include the case $l = 1$ here too, if, in case of $\mathsf{d} = 0$, only one payload block has been encrypted during parallel processing.

Tag Generation

In NORX, the generation of an authentication tag T works as follows: first, the domain separation constant 08 is included into s_{15}. Then s is updated *twice* with F^{r}. Finally, the rate words $s_0, \ldots, s_{r/\mathsf{w}-1}$ are converted back to a bit string using $\mathsf{to_str}_{\mathsf{w}}$ and the t least significant bits are extracted from the result and set as T. Algorithm 16 shows the pseudo code of the tag generation operation.

Algorithm 16: $\mathsf{generate_tag}(s)$

Inputs:
$s \in \mathbb{Z}_{2^{\mathsf{w}}}^{16}$
Outputs:
$s \in \mathbb{Z}_{2^{\mathsf{w}}}^{16}$, $T \in \mathbb{F}_2^{\mathsf{t}}$
Algorithm:
1. $s_{15} \leftarrow s_{15} \oplus 08$
2. $s \leftarrow \mathsf{F}^{\mathsf{r}}(\mathsf{F}^{\mathsf{r}}(s))$
3. $T \leftarrow \lfloor \mathsf{to_str}_{\mathsf{w}}((s_0, \ldots, s_{r/w-1})) \rfloor_{\mathsf{t}}$
4. **return** s, T

4.2.6 Decryption Mode

NORX decryption can process messages M of the form $M = A \parallel C \parallel B$, where A denotes a *header*, C an *encrypted payload*, and B a *trailer*. Like in encryption, associated data A and B as well as the payload C can be potentially empty. Decryption additionally takes an authentication tag T as input.

Structure

NORX decryption has a very similar structure to encryption, see Section 4.2.5. Refer to Algorithm 17 for a summary of the steps executed during decryption.

Padding

Padding is identical to that of encryption, i.e. the multi-rate padding is used.

Domain Separation

The domain separation constants and their application are the same as in encryption.

Initialisation

Initialisation is identical to that of encryption.

Algorithm 17: norx_decrypt(K, N, A, C, B, T)

Inputs:
$K \in \mathbb{F}_2^{4\mathsf{w}}$, $N \in \mathbb{F}_2^{2\mathsf{w}}$, $A \in \mathbb{F}_2^*$, $C \in \mathbb{F}_2^*$, $B \in \mathbb{F}_2^*$, $T \in \mathbb{F}_2^{\mathsf{t}}$
Outputs:
$P \in \mathbb{F}_2^*$, with $|P| = |C|$, or $\perp$
Algorithm:
1. $s \leftarrow \mathsf{initialise}(K, N)$
2. $s \leftarrow \mathsf{absorb_data}(s, A, \mathsf{01})$
3. $s \leftarrow \mathsf{branch}(s, |\mathsf{pad}_r(P)|)$
4. $s, P \leftarrow \mathsf{decrypt_data}(s, C, \mathsf{02})$
5. $s \leftarrow \mathsf{merge}(s)$
6. $s \leftarrow \mathsf{absorb_data}(s, B, \mathsf{04})$
7. $s, T' \leftarrow \mathsf{generate_tag}(s)$
8. **if** $T \neq T'$ **then**
9. **return** $\perp$
10. **end**
11. **return** P

Message Processing

Message processing in decryption is similar to that in encryption: header and trailer processing are identical, but payload processing is different. The latter is done using the functions decrypt_data and decrypt_block as shown in Algorithms 18 and 19. Like in encryption as many bits are extracted and written to P as unpadded encrypted payload bits.

Tag Generation

Tag generation is identical to that in encryption.

Tag Verification

Tag verification consists of comparing the *received tag* T to the *generated tag* T'. If $T = T'$, tag verification succeeds and the decrypted payload is returned; otherwise tag verification fails, the decrypted payload is discarded and an error $\perp$ is returned. See Algorithm 17 for more information. Implementations of tag verification should satisfy the following requirements:

- Tag verification should not leak information on the (relative) values of the compared bit strings. In particular, tag verification should be implemented in constant time, so that a comparison of identical strings and distinct strings takes the same time.
- Decrypted data should not be returned to the user if tag verification fails and ideally should be erased securely from any temporary memory.

Algorithm 18: decrypt_data(s, X, v)

Inputs:
$s \in \mathbb{Z}_{2^{\mathsf{w}}}^{16l}$, with $l \geq 1$, $X \in \mathbb{F}_2^*$, $v \in \mathbb{Z}_{2^{\mathsf{s}}}$
Outputs:
$s \in \mathbb{Z}_{2^{\mathsf{w}}}^{16l}$, $P \in \mathbb{F}_2^*$, with $|X| = |P|$
Algorithm:
1. $P \leftarrow \varepsilon$
2. **if** $|X| > 0$ **then**
3. $\quad X' \leftarrow \mathsf{pad}_r(X)$
4. $\quad m \leftarrow \mathsf{d}$
5. $\quad$ **if** $\mathsf{d} = 0$ **then**
6. $\quad\quad m \leftarrow |X'|_r$
7. $\quad$ **end**
8. $\quad$ **for** $i \in \{0, \ldots, |X'|_r - 1\}$ **do**
9. $\quad\quad j = i \bmod m$
10. $\quad\quad s' \leftarrow (s_{16j}, \ldots, s_{16j+15})$
11. $\quad\quad s', Y \leftarrow \mathsf{decrypt_block}(s', X'_i, v)$
12. $\quad\quad (s_{16j}, \ldots, s_{16j+15}) \leftarrow s'$
13. $\quad\quad P \leftarrow P \parallel Y$
14. $\quad$ **end**
15. $\quad P \leftarrow \lfloor P \rfloor_{|X|}$
16. **end**
17. **return** s, P

Algorithm 19: decrypt_block(s, X, v)

Inputs:
$s \in \mathbb{Z}_{2^{\mathsf{w}}}^{16}$, $X \in \mathbb{F}_2^r$, $v \in \mathbb{Z}_{2^{\mathsf{s}}}$
Outputs:
$s \in \mathbb{Z}_{2^{\mathsf{w}}}^{16}$, $Y \in \mathbb{F}_2^r$
Algorithm:
1. $x \leftarrow \mathsf{to_int}_{\mathsf{w}}(X)$
2. $y \leftarrow \mathsf{to_int}_{\mathsf{w}}(0^r)$
3. $s_{15} \leftarrow s_{15} \oplus v$
4. $s \leftarrow \mathsf{F}^{\mathsf{r}}(s)$
5. **for** $i \in \{0, \ldots, r/\mathsf{w} - 1\}$ **do**
6. $\quad y_i \leftarrow s_i \oplus x_i$
7. $\quad s_i \leftarrow x_i$
8. **end**
9. $Y \leftarrow \mathsf{to_str}_{\mathsf{w}}(y)$
10. **return** s, Y

4.2.7 Datagrams

Many issues with encryption interoperability are due to ad hoc ways to represent and transport cryptograms and the associated data. For example, nonces or initialisation vectors (IVs) are sometimes prepended to the ciphertext, sometimes appended, or sent separately. We thus specify datagrams that can be integrated in a protocol stack, encapsulating the ciphertext as a payload. Using a standardized encoding simplifies the transmission of NORX cryptograms across different APIs and reduces the risk of insecure or suboptimal encodings. We specify two distinct types of datagrams, depending on whether the NORX parameters are fixed or need to be signalled in the datagram header.

Fixed Parameters

With *fixed parameters* shared by the parties (for example through the application using NORX), there is no need to include the parameters in the *header of the datagram*[3]. The datagram for fixed parameters thus only needs to contain N, A, C, B, and T, as well as information to parse those elements.

We encode the byte length of A and B by 16 bits, allowing for headers and trailers of

[3]The header referred to is that of the datagram specified, not that of the data processed by the NORX instance.

up to 64 KiB, a large enough value for most real applications. The byte length of the encrypted payload is encoded on 32 bits for NORX32 and on 64 bits for NORX64 which translates to a maximum payload size of 4 GiB and 16 EiB, respectively[4]. Similarly to frame check sequences in data link protocols, the tag is added as a *trailer of the datagram* specified. The header, encrypted payload, and trailer of the underlying protocol are viewed as the *payload of the datagram*. The default tag length being a constant value of the NORX instance, it needs not be signalled.

Table 17 shows the fixed-parameters datagrams for NORX32 and NORX64. The length of the datagram header is 28 bytes for NORX64 and 16 bytes for NORX32.

Note that the CAESAR API (according to the final call, see [84]) receives the nonce and the associated data in two separate buffers, but the tag is included in the ciphertext buffer.

Variable Parameters

With *variable parameters*, the datagram needs to signal the values of w, r, and d. The header is thus extended to encode those values, as specified in Table 18. To minimize bandwidth, w is encoded on one bit, supporting the two choices 32-bit ($w = 0$) and 64-bit ($w = 1$), r on 7 bits (with the MSB fixed at 0, i.e. supporting up to 63 rounds), and d on 8 bits (supporting parallelization degree up to 255). The datagram header is thus only 2 bytes longer than the header for fixed parameters.

4.3 Security Goals

We expect NORX with $r \geq 4$ to provide the maximum security for any AEAD scheme with the same interface (input and output types and lengths). The following requirements should be satisfied in order to use NORX securely:

1. **Unique Nonces.** Each key and nonce pair should not be used to process more than one message.

2. **Abort on Verification Failure.** If the tag verification fails, only an error is returned. In particular, the decrypted plaintext and the wrong authentication tag must not be given as an output and should be erased from memory in a safe way.

In the simplest case, the first requirement can be realised by implementing the nonce as a counter, which is increased for each new message processed under a given, fixed key. If the key is changed, the counter can be reset to start all over again.

[4]Note that NORX is capable of (safely) processing much larger data sizes, those are just the maximum values when our proposed datagrams are used.

Table 17: NORX datagrams for fixed parameters, all offsets are in bytes.

NORX32				
Offset	**0**	**1**	**2**	**3**
0 **4**	Nonce N			
8	Header byte length $\|A\|$		Trailer byte length $\|B\|$	
12	Encrypted payload byte length $\|C\|$			
16 ... **??**	Header A			
?? ... **??**	Encrypted payload C			
?? ... **??**	Trailer B			
?? ... **??**	Tag T			

NORX64				
Offset	**0**	**1**	**2**	**3**
0 **4** **8** **12**	Nonce N			
16	Header byte length $\|A\|$		Trailer byte length $\|B\|$	
20 **24**	Encrypted payload byte length $\|C\|$			
28 ... **??**	Header A			
?? ... **??**	Encrypted payload C			
?? ... **??**	Trailer B			
?? ... **??**	Tag T			

Table 18: NORX datagrams for variable parameters, all offsets are in bytes.

NORX32				
Offset	**0**	**1**	**2**	**3**
0 **4**	Nonce N			
8	Header byte length $\|A\|$		Trailer byte length $\|B\|$	
12	Encrypted payload byte length $\|C\|$			
16	w(1) ‖ r(7)	d		
20 ... **??**	Header A			
?? ... **??**	Encrypted payload C			
?? ... **??**	Trailer B			
?? ... **??**	Tag T			

NORX64				
Offset	**0**	**1**	**2**	**3**
0 **4** **8** **12**	Nonce N			
16	Header byte length $\|A\|$		Trailer byte length $\|B\|$	
20 **24**	Encrypted payload byte length $\|C\|$			
28	w(1) ‖ r(7)	d		
32 ... **??**	Header A			
?? ... **??**	Encrypted payload C			
?? ... **??**	Trailer B			
?? ... **??**	Tag T			

We do not make any claim regarding attackers using "related keys", "known keys", "chosen keys", etc. We also exclude from the claims below models where information is "leaked" on the internal state or key.

The security of NORX is limited by the key length (128 or 256 bits) and by the tag length (128 or 256 bits). Plaintext confidentiality should thus have the order of 128 or 256 bits of security. The same level of security should hold for integrity of the plaintext or of associated data based on the fact that an attacker trying 2^n tags will succeed with probability 2^{n-256} and $n < 256$ to forge a tag. In particular, recovery of a k-bit NORX key should require resources ("computations", energy, etc.) comparable to those required to recover the key of an ideal k-bit key cipher. Table 19 summarizes the security goals of NORX.

Table 19: Overview on the expected security levels (in bits).

Security goal	NORX32	NORX64
Plaintext confidentiality	128	256
Plaintext integrity	128	256
Associated data integrity	128	256
Public message number integrity	128	256

Usage Exponent

NORX restricts the number of messages processed with a given key: in [43] the *usage exponent* e is defined as the value such that the implementation imposes an upper limit of 2^e uses to a given key. In NORX we set it to $e_{64} = 128$ for 64-bit and $e_{32} = 64$ for 32-bit, which corresponds in both cases to the size of the nonce. NORX has capacities of $c_{64} = 384$ (64-bit) and $c_{32} = 192$ (32-bit). As a consequence, security levels of at least $c_{64} - e_{64} - 1 = 384 - 128 - 1 = 255$ bits for NORX64 and $c_{32} - e_{32} - 1 = 192 - 64 - 1 = 127$ bit for NORX32 are expected, see [43].

Security Bounds for the Mode of Operation

In [141] the generic security bounds in the ideal permutation model for the NORX mode of operation are analysed, together with some of the other sponge-based CAESAR candidates. We give at this point only a brief overview on the main results and refer to the above work for more details. Moreover, see Section 1.3 for the basic notions of provable security.

Let $\Pi = (\mathcal{E}, \mathcal{D})$ denote NORX, with encryption function $\mathcal{E}$, decryption function $\mathcal{D}$, which are both assumed to be based on an ideal underlying permutation p. Further, let r, c, b, k, and t denote sizes for rate, capacity, state, key, and tag of NORX, let d be the

parallelism degree of the scheme, let e be Euler's number, q_p the number of permutation queries, $q_{\mathcal{E}}$ the number of encryption queries of total length $\lambda_{\mathcal{E}}$, and let $\sigma_{\mathcal{E}}$ be specified as follows:

$$\sigma_{\mathcal{E}} := \sum_{j=1}^{q_{\mathcal{E}}} \sigma_{\mathcal{E},j} \leq \begin{cases} 2\lambda_{\mathcal{E}} + 4q_{\mathcal{E}}, & \text{if } \mathsf{d} = 0 \\ \lambda_{\mathcal{E}} + 3q_{\mathcal{E}}, & \text{if } \mathsf{d} = 1 \\ \lambda_{\mathcal{E}} + (\mathsf{d}+4)q_{\mathcal{E}}, & \text{if } \mathsf{d} > 1 \ . \end{cases}$$

The values $q_{\mathcal{D}}$, $\lambda_{\mathcal{D}}$ and $\sigma_{\mathcal{D}}$ for decryption $\mathcal{D}$ are specified analogously. Furthermore, it is assumed that adversaries are nonce-respecting for encryption, i.e. encrypting different messages with the same nonce-key pair is not permitted. This constraint is not necessary for decryption though. Then the following two propositions hold for the confidentiality and authenticity security bounds of NORX.

Proposition 1. *An adversary who aims to break the confidentiality of the NORX mode of operation based on an ideal permutation p is able to achieve the following maximal advantage:*

$$\mathbf{Adv}_{\Pi}^{\mathsf{priv}}(q_p, q_{\mathcal{E}}, \lambda_{\mathcal{E}}) \leq \frac{3(q_p + \sigma_{\mathcal{E}})^2}{2^{b+1}} + \left(\frac{8eq_p\sigma_{\mathcal{E}}}{2^b}\right)^{1/2} + \frac{rq_p}{2^c} + \frac{q_p + \sigma_{\mathcal{E}}}{2^k} \ .$$

Proof. See [141, proof of Theorem 1]. □

Proposition 2. *An adversary who aims to break the integrity of the NORX mode of operation based on an ideal permutation p is able to achieve the following maximal advantage:*

$$\mathbf{Adv}_{\Pi}^{\mathsf{auth}}(q_p, q_{\mathcal{E}}, \lambda_{\mathcal{E}}, q_{\mathcal{D}}, \lambda_{\mathcal{D}}) \leq \frac{(q_p + \sigma_{\mathcal{E}} + \sigma_{\mathcal{D}})^2}{2^b} + \left(\frac{8eq_p\sigma_{\mathcal{E}}}{2^b}\right)^{1/2} + \frac{q_{\mathcal{D}}}{2^{\mathsf{t}}} + \\ + \frac{q_p + \sigma_{\mathcal{E}} + \sigma_{\mathcal{D}}}{2^k} + \frac{rq_p + (q_p + \sigma_{\mathcal{E}} + \sigma_{\mathcal{D}})\sigma_{\mathcal{D}}}{2^c} \ .$$

Proof. See [141, proof of Theorem 2]. □

In summary, the above two results show that the NORX mode of operation roughly achieves security levels for authenticity and confidentiality of

$$\mathsf{min}\{2^{b/2}, 2^c, 2^k\}$$

(recall that $k = \mathsf{t}$), for all $0 \leq \mathsf{d} \leq 255$, assuming an ideal underlying permutation p and a nonce-respecting adversary. Intuitively spoken, NORX offers authenticity as long as it offers privacy. In particular, since $b/2 > c > k$, the generic security level of NORX is determined by the size of the secret key, as one would expect.

In [7] another approach to the security of keyed sponge constructions is presented which generalises the results of [141]. However, for the concrete case of NORX, the two results on the security bounds basically concur. Additionally, this work also evaluates security bounds for the multi-key setting, i.e. the scenario where an adversary simultaneously attacks instances of an authenticated encryption scheme that use different keys. It is proven that an adversary in the multi-key setting basically has no advantage over an adversary that is restricted to the single-key setting. These results also apply to NORX.

4.4 Features

NORX provides several features desirable for practical applications and offers a couple of important advantages over AES-GCM [196]. First, we list these characteristics in detail, then give a justification of our recommended parameter sets, and finally present our performance results.

4.4.1 List of Characteristics

High Security. NORX supports 128- and 256-bit keys and authentication tags of arbitrary size, thanks to its duplex construction. The core permutation of NORX was designed and evaluated to be cryptographically strong. The minimal number of $R = 8$ rounds for initialisation / finalisation, i.e. eight interleaved applications of col and diag operations each (16 steps in total), and of $R = 4$ rounds for the data processing part, i.e. 4 interleaved applications of col and diag operations each (8 steps in total), should ensure a high security margin against cryptanalytic attacks. Furthermore, large internal states of 512 and 1024 bits and the duplex construction offer protection against generic attacks.

Efficiency. NORX was designed with 64-bit processors in mind, but is also compatible with smaller architectures like 8- to 32-bit platforms. Software implementations of NORX are able to take advantage of multi-core processors, due to the parallel duplex construction, and specialised instruction sets like the Intel advanced vector instructions (AVX, AVX2, AVX-512) [135] or the ARM advanced SIMD extensions (NEON) [9]. Optimising an algorithm towards those vector instructions is especially important in order to keep-up performance-wise with AES-based ciphers, like AES-GCM, which have huge speed advantages when special AES instructions such as AES-NI [122] are available. Moreover, state sizes of 512 and 1024 bits make NORX very cache-friendly. Hardware implementations benefit from hardware-friendly operations, next to the arbitrary parallelism degree for payload processing which results in highly competitive hardware performance of NORX.

Simplicity. The core algorithm iterates a simple round function and can be implemented by translating our pseudocode into the programming language used: NORX requires no

S-boxes, no Galois field operations, and no integer arithmetic; `AND`, `XOR`, and shifts are the only instructions required. This simplifies security analysis and the task of implementing the cipher.

High Key Agility. NORX requires no key expansion when setting up a new key, in contrast to many block-cipher based schemes, like AES-GCM. Switching the secret key is therefore very cheap. As an additional benefit, there are also no hidden costs of loading precomputed expanded keys from DRAM into L1 cache.

Adjustable Tag Sizes. The NORX family allows tag sizes of up to 10w bits, with a default of 4w bits for our proposed instances. Thanks to the duplex construction, tag sizes can be easily adapted to the demands of any given application.

Simple Integration. NORX can be easily integrated into a protocol stack, as it supports flexible processing of arbitrary datagrams: any header and trailer are authenticated (and left unencrypted) and the payload is both encrypted and authenticated.

Interoperability. Dedicated datagrams encode parameters of the cipher and encapsulate the protected data. This aims to increase interoperability across implementations.

Single Pass. Encryption and authentication as well as decryption and tag verification are done in a single pass of the algorithm.

Online. NORX supports encryption and decryption of data streams, i.e. the size of processed data needs not to be known in advance.

High Data Processing Volume. NORX allows to process very large data sizes from a single key-nonce pair. The usage exponent, see Section 4.3, theoretically limits the number of calls to the core permutation to values of 2^{64} (NORX32) and 2^{128} (NORX64), respectively. This translates to data sizes which are orders of magnitude beyond everything relevant for current and future real-world applications. In particular, these values are a lot higher than the maximum of 2^{32} calls to the authenticated encryption function of AES-GCM which could be easily reached nowadays in real-world systems.

Minimal Overhead. Payload encryption is non-expanding, i.e. the ciphertext has the same length as the plaintext. The authentication tag has a length of 16 or 32 bytes depending on the concrete instance of NORX.

Robustness Against Timing Attacks. By avoiding data-dependent table look-ups, like S-boxes, the goal to harden implementations of NORX against timing-attacks [35] should be comparably easy to achieve, since no special implementations, like bit-sliced S-boxes [3, 90] for constant-time table-lookups, are required.

Moderate Misuse Resistance. NORX retains its security even if nonces are reused as long as it can be guaranteed that header data is unique[5]. For comparison, nonce reuse in AES-GCM is a major security issue, allowing an attacker to recover the secret key [136].

Autonomy. NORX requires no external primitive.

Diversity. The cipher does not depend on AES instructions, thereby adding to the diversity among cryptographic algorithms.

Extensibility. NORX can be easily extended to support additional features, such as secret message numbers, sessions, or forward-secrecy without losing its security guarantees, thanks to the flexibility of the duplex construction and a simple, yet powerful domain separation scheme.

4.4.2 Recommended Parameter Sets

The recommended parameter sets are listed in Table 14. We consider NORX32-4-1 and NORX64-4-1 as the standard instances for the respective word sizes of 32 and 64 bit. These configurations offer a good balance between performance and security. We recommend NORX32-4-1 for low resource applications on 8- to 32-bit platforms and NORX64-4-1 for software implementations on modern 64-bit CPUs or standard hardware implementations. Applications that require a higher security margin and where performance has less priority are advised to use the instances NORX32-6-1 and NORX64-6-1. For use cases where very high data throughput is necessary, we recommend NORX64-4-4, which allows payload encryption on four parallel lanes, thus enabling very high data processing speeds.

4.4.3 Performance

NORX was designed to perform well across both software and hardware. This section details our implementations and performance results.

[5] Nevertheless, the designers discourage this approach, and recommend that nonce freshness should be ensured by all means.

Generalities

In this part, we analyse some general performance-relevant properties of NORX, like number of operations in G and F^r, parallelism degree, and upper bounds for the speed of NORX on different platforms.

Number of Operations. Table 20 shows the number of operations required for the NORX core functions. We omit the overhead of initialisation, integration of parameters, domain separation constants, padding messages, and so on, as those costs are negligible compared to that of the core permutation F^r.

Table 20: Overview on the number of operations of the NORX functions.

Function	#XOR	#AND	#Shifts	#Rotations	Total
G	12	4	4	4	24
F	96	32	32	32	192
F^4	384	128	128	128	768
F^6	576	192	192	192	1152
F^8	768	256	256	256	1536
F^{12}	1152	384	384	384	2304

Memory. NORX32 and NORX64 require at least 16 and 32 bytes to be stored in ROM for the initialisation constants[6]. To store all initialisation constants 40 and 80 bytes of ROM are necessary.

Processing a message in NORX requires enough RAM to store the internal state, i.e., 64 bytes in NORX32 and 128 bytes in NORX64. The data being processed need not be in memory for more than 1 byte at a time. In practice, however, it is preferable to process blocks of 40 respectively 80 bytes at a time.

Parallelism. The core permutation F of NORX has a natural parallelism of 4 independent G applications. Additionally, NORX allows for greater parallelism levels using multiple lanes. Using the $\mathsf{d} = 0$ mode (see part on message processing in Section 4.2.5), the internal parallelism level of NORX is effectively unbounded for long enough messages.

Software

NORX is easily implemented for 32-bit and 64-bit processors, as it works on 32- and 64-bit words and uses only word-based operations (XOR, AND, shifts and rotations). The

[6]The ten constants can be generated on the fly from the four basic constants $u_0, \ldots, u_3$, see initialisation in Section 4.2.5.

specification can be translated directly to code and requires no specific technique such as look-up tables or bitslicing. The core of NORX essentially consists of repeated usage of the G function, which allows simple and compact implementations (e.g., by having only one copy of the G code).

Furthermore, constant-time implementations of NORX are straightforward to write, due to the absence of secret-dependent instructions or branchings.

Bit Interleaving. While NORX's lack of integer addition avoids dealing with carry chains, the implementer may still have to perform full-word rotations and shifts in words wider than the natural CPU word size. In 8-bit processors, some of this burden is alleviated by 2 out of 4 rotations being multiples of 8. However, this is only a half-measure.

Instead, the implementer can employ the *bit interleaving* technique presented in [50]. This technique consists of splitting an n-bit word w into $s = n/m$ m-bit words b_i, with $b_{ij} = w_{i+jn/m}$. A rotation by r in this representation can be performed by rotating each b_i by $\lfloor r/w \rfloor + 1$ if $i + r \bmod m < r$, $\lfloor r/w \rfloor$ otherwise, and moving b_i to $b_{i+r \bmod m}$. Rotations by 1 or $n-1$ are particularly attractive, since they result in a single m-bit rotation. For example, consider implementing NORX64 on a 32-bit CPU. Each state word w will be split into the two words b_0 and b_1. To rotate by r:

- If $r \bmod 2 = 0$, rotate both b_0 and b_1 by $\lfloor r/2 \rfloor$;
- If $r \bmod 2 = 1$, rotate b_1 by $\lfloor r/2 \rfloor + 1$, b_0 by $\lfloor r/2 \rfloor$, and swap them.

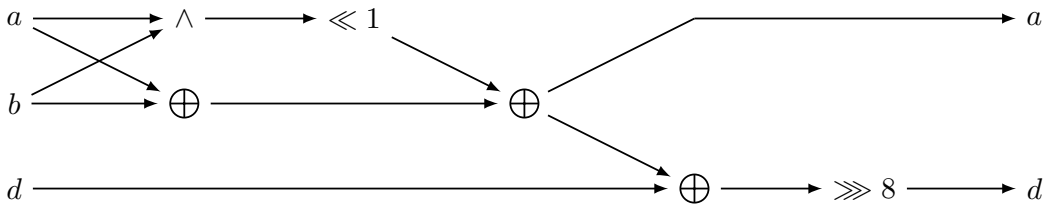
Conversion between representations can be performed in logarithmic time using bit "zip" and "unzip" operations [11].

Avoiding Latency. One drawback of G is that it has little instruction parallelism. In architectures where one is limited by the latency of the G function, an implementer can trade a few extra instructions by reduced latency:

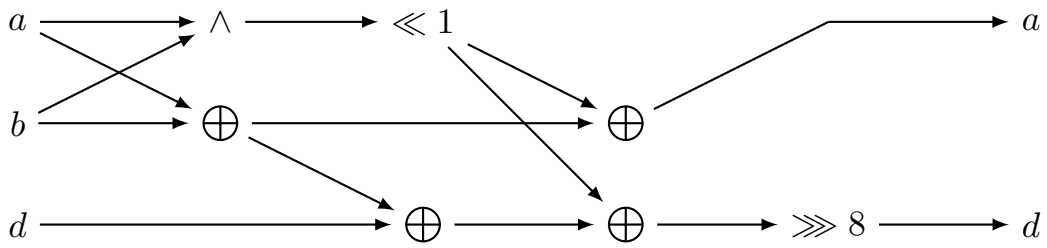
$$\begin{aligned} t_0 &\longleftarrow a \oplus b \\ t_1 &\longleftarrow a \wedge b \\ t_1 &\longleftarrow t_1 \ll 1 \\ a &\longleftarrow t_0 \oplus t_1 \\ d &\longleftarrow d \oplus t_0 \\ d &\longleftarrow d \oplus t_1 \\ d &\longleftarrow d \ggg r_0 \ . \end{aligned}$$

This tweak saves up to 1 cycle per instruction sequence, of which there are 4 per G, at the cost of 1 extra instruction (cf. Figure 36). In a sufficiently parallel architecture, this can

save at least $4 \cdot 2 \cdot \mathsf{r}$ cycles, which translates to $6.4\mathsf{r}/\mathsf{w}$ cycles per byte saved overall. In our measurements, this translated to a performance improvement of NORX from 0.4 to 0.7 cycles per byte, depending on the target architecture, word size, and number of rounds.



(a) Naïve implementation of the G instruction sequence.



(b) Latency-oriented version of the G instruction sequence.

Figure 36: Improving the latency of G.

Vectorization. NORX lends itself quite well to implementations taking advantage of SIMD extensions present in modern processors, such as AVX or NEON. The typical vectorized implementation of NORX, when $\mathsf{d} = 1$, works in full rows of the 4×4 state and computes whole column and diagonal steps of F in parallel.

Results. We wrote portable C reference implementations for both the 32- and 64-bit versions of NORX, as well as optimized versions for CPUs supporting AVX and AVX2 and for NEON-enabled ARMs. Table 21 shows speed measurements on various platforms for messages with varying lengths. The listed CPU frequencies are nominal ones, i.e. without dynamic overclocking features like Turbo Boost which improves the accuracy of measurements. Furthermore, we listed only those platform-compiler combinations that achieved the highest speeds. Common compiler flags used on all platforms were `-O3 -std=c89 -Wall -pedantic -Wno-long-long`. Additionally, some platform and/or implementation dependent compiler flags were set for each benchmark, like `-march=armv7-a -mcpu=cortex-a8 -mpfu=neon` for measurements of the NEON

implementations on the BeagleBone Black. All software benchmarks are measured in *cycles per byte* (cpb).

On the Intel platforms the top speed of NORX (for $\mathsf{d} = 1$), in terms of bytes per second, was achieved by an AVX2 implementation of NORX64-4-1 on a Haswell CPU as listed in Table 21. It achieves a throughput of about 1.39 GiBps (2.51 cycles per byte at 3.5 GHz). The top speeds on the Sandy Bridge and Ivy Bridge architectures are reached by an AVX implementation of NORX64-4-1. They run at speeds of 3.28 and 3.37 cycles per byte which correspond to throughputs of 609 MiBps and 593 MiBps, respectively, both for a CPU frequency of 2.0 GHz.

On the ARM platforms the highest speed was achieved by the reference implementation of NORX64-4-1 on the Apple A7. It runs at 4.07 cycles per byte, which translates to a throughput of 343 MiBps for a frequency of 1.4 GHz. At a first glance, it might be surprising that the reference implementation outperforms the NEON-optimised variant, however, at a closer look, this "anomaly" can be traced back to the special design of the Apple chip. The A7 has been the first 64-bit ARM processor available to consumers. It has 4 integer ALUs, is capable of executing 4 integer and 2 floating point additions per cycle, i.e. up to 6 operations per cycle, and has a reorder buffer of 192 micro-operations which are quite impressive numbers when compared to the Intel architectures since older ARM chips usually had smaller reorder buffers and fewer ALUs. For example, the Ivy Bridge and Haswell chips are capable to execute up to 6 and 8 operations per cycle and have reorder buffers of 168 and 192 micro-operations, respectively. We presume that the high number of integer ALUs and free 1-bit shift operations on the ARM architecture are the main reason why the NORX reference implementation outperforms its NEON-optimised counterpart on the Apple A7. On the Cortex-A8 and -A9, the top speeds are achieved by our NEON implementation of NORX64-4-1, running at 8.96 and 8.94 cycles per byte. Hence, throughputs of 111 MiBps and 190 MiBps are reached for frequencies of 1.0 GHz (Cortex-A8) and 1.7 GHz (Cortex-A9).

For comparison, state-of-the-art NEON implementations of Salsa20 and Poly1305 [37] achieve 5.60 cycles per byte for encryption and 2.30 cycles per byte for authentication of a message, as reported in [42]. Hence, encrypting and authenticating a message requires about 7.90 cycles per byte in total. The benchmarks there were done on a Cortex-A8 having a frequency of 800 MHz. With a difference of about 1 cycle per byte, the speeds of NORX on our Cortex-A8 is a little worse but still comparable to those of [42]. However, we only need one pass over the data whereas the above implementation requires two, one for encryption (using Salsa20) and one for authentication (using Poly1305). Moreover, it is important to note that the code in [42] was written directly in assembly language using the `qhasm` framework [39], whereas our NORX code was written only partially in assembly language. An assembly implementation has the advantage that no C compiler is required, which eliminates problems like bad code scheduling of the latter. A complete assembly language implementation of NORX64-4-1 would probably bring its speed closer to the above 7.90 cycles per byte of Salsa20-Poly1305, due to the circumvention of possible bad

C compiler behaviour.

The overhead for short messages (≤ 64 bytes), as depicted in Table 21, is mainly due to the additional initialisation and finalisation rounds (see Figure 32). However, the cost per byte quickly decreases and stabilizes for messages larger than about 1 KiB. Figure 37 presents a visualisation of the performance measurements on the different platforms.

Note that the speed between reference and optimized implementations differs by a factor of less than two, suggesting that straightforward and portable implementations will provide sufficient performance for most applications. Such consistent performance reduces development costs and improves interoperability and predictability.

Hardware

Hardware architectures of NORX are efficient and easy to design from the specification: vertical and parallel folding are naturally derived from the iterated and parallel structure of NORX. The cipher benefits from the hardware-friendliness of the function G, which requires only bitwise logical `AND`, `XOR`, and bit shifts, and the iterated usage of G inside the core permutation of NORX.

A team around Gürkaynak at ETH Zürich designed and taped out an ASIC, called CronorX [152], supporting parameters $\mathsf{w} \in \{32, 64\}$, $\mathsf{r} \in \{2, \ldots, 16\}$, and $\mathsf{d} = 1$. It was synthesized with the Synopsys Design Compiler using 180 nm UMC technology. The implementation was targeted at high data throughput. The requirements in area amounted to about 59 kGE. Simulations for NORX64-4-1 report a throughput of about 10 Gbps (1.2 GiBps), at a frequency of 125 MHz. A picture of the chip is shown in Figure 38.

A more thorough evaluation of all hardware aspects of NORX is planned for the future. Due to the similarity of NORX to ChaCha and the fact that NORX has only little overhead compared to a blank stream cipher, we expect similar results to those of ChaCha as presented in [129].

NORX versus AES-GCM

Since AES-GCM is basically the current de-facto standard for authenticated encryption and the CAESAR candidates are meant to replace AES-GCM in the long run, we present in the following a short performance comparison of NORX to AES-GCM. More precisely, we compare 128- and 256-bit variants of the two AEAD schemes and use our standard instances NORX32-4-1 and NORX64-4-1 during our investigations.

Software. AES-GCM achieves very high speeds on modern x86 processors when the *AES New Instructions* (AES-NI) extensions are available. Gueron [121] reports, for the 128-, 192-, and 256-bit key variants, running times of 1.03 cpb, 1.17 cpb, and 1.31 cpb on a Haswell processor. The situation is different when AES-NI is not available which is the case for the majority of platforms. For example, in [149] two different implementations

Table 21: Software performance of NORX in cycles per byte.

data length [byte]		long	4096	1536	576	64	8
Samsung Exynos 4412 Prime (Cortex-A9) at 1.7 GHz							
NORX32-4-1	Ref	21.57	22.86	24.94	30.50	97.94	663.75
	NEON	10.57	11.41	12.77	16.40	61.73	434.88
NORX64-4-1	Ref	26.68	28.49	32.20	42.62	152.16	1218.75
	NEON	8.94	9.94	11.79	16.79	73.70	584.50
BeagleBone Black Rev B (Cortex-A8) at 1.0 GHz							
NORX32-4-1	Ref	19.76	21.21	23.53	29.74	106.02	744.00
	NEON	10.50	11.57	13.30	17.92	75.62	550.12
NORX64-4-1	Ref	25.82	27.82	31.83	42.79	161.61	1286.12
	NEON	8.96	10.15	12.32	18.15	84.80	673.88
Apple A7 (64-bit ARMv8) at 1.4 GHz							
NORX32-4-1	Ref	7.98	8.32	8.82	10.20	60.55	395.75
	NEON	11.90	12.33	13.02	14.90	87.23	562.50
NORX64-4-1	Ref	4.07	4.34	4.91	6.29	50.78	401.00
	NEON	7.34	7.80	8.76	11.21	86.58	703.12
Intel Core i7-2630QM at 2.0 GHz							
NORX64-6-1	Ref	7.69	8.14	9.08	11.54	37.75	304.00
	AVX	4.94	5.24	5.90	7.52	24.81	198.00
NORX64-4-1	Ref	5.28	5.59	6.24	7.94	26.00	208.00
	AVX	3.28	3.49	3.91	5.03	16.69	133.50
Intel Core i7-3667U at 2.0 GHz							
NORX64-6-1	Ref	7.04	7.46	8.32	10.59	34.87	371.50
	AVX	5.04	5.37	6.03	7.71	25.44	276.00
NORX64-4-1	Ref	4.92	5, 24	5.86	7.43	24.93	310.00
	AVX	3.37	3.59	4.01	5.16	17.18	218.00
Intel Core i7-4770K at 3.5 GHz							
NORX64-6-1	Ref	6.63	7.00	7.77	9.85	32.12	256.50
	AVX2	3.73	3.98	4.47	5.71	19.19	153.00
NORX64-4-1	Ref	4.50	4.76	5.27	6.71	22.06	176.00
	AVX2	2.51	2.66	3.01	3.83	13.06	104.00

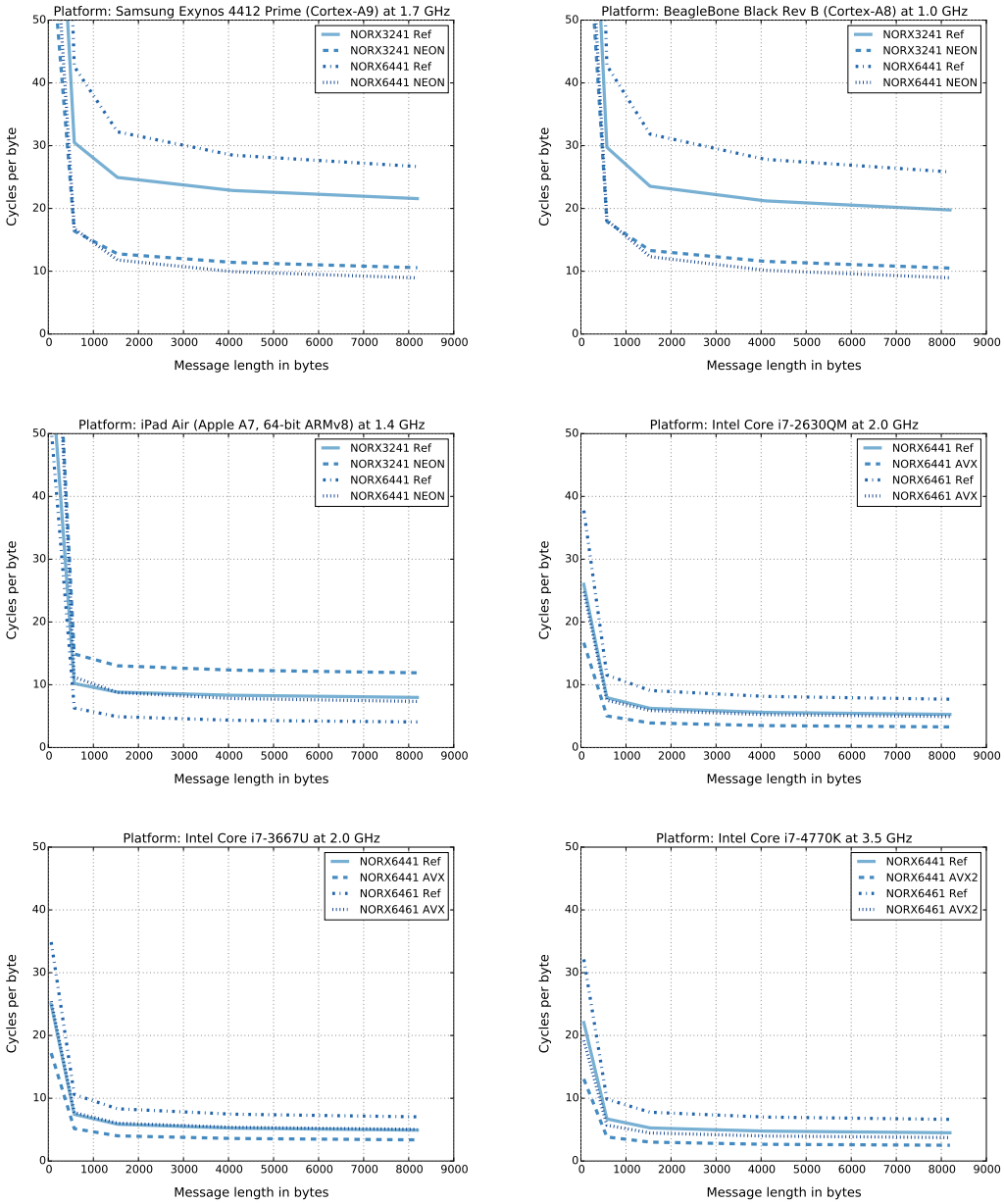


Figure 37: Visualisations for the software performance measurements of NORX.

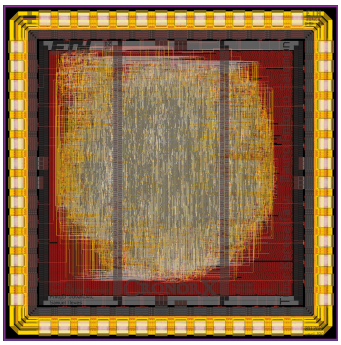


Figure 38: The CronorX chip.

of AES128-GCM are analysed on a Nehalem processor, one constant-time version that runs at 20.29 cpb and one tuned towards performance but vulnerable to timing-attacks that runs at 10.12 cpb. On ARM processors, where AES-NI is not available as well, the picture is also a very different one. On the Cortex-A8 chip for example, speeds of 50.8 cpb (standard) and 38.6 cpb (NEON) are reported for AES128-GCM in [171] and [93], respectively.

For better comparisons, we ran our own benchmarks of AES128-GCM and AES256-GCM using the implementations of the OpenSSL library (version 1.0.1j) on the Ivy Bridge and Cortex-A8 chips as listed in Table 21. A summary of our results is given in Figure 39.

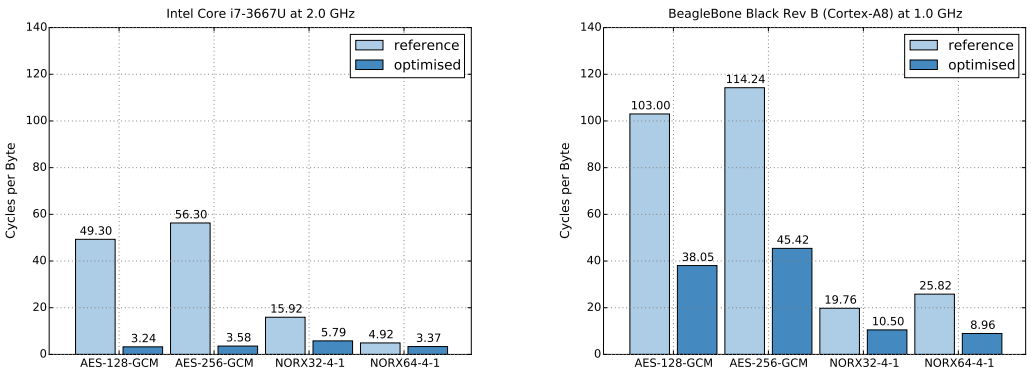


Figure 39: NORX versus AES-GCM (OpenSSL 1.0.1j).

On the Ivy Bridge and with activated hardware support we measured speeds of about 3.24 and 3.58 cpb for AES128-GCM and AES256-GCM, respectively. In comparison, AVX versions of NORX32-4-1 and NORX64-4-1 run at 5.79 and 3.37 cpb which corresponds to speed differences by factors of 0.55 and 1.06. Note that the speed of NORX64-4-1

is already on par with AES256-GCM and AES128-GCM is only slightly faster. The advantage shifts entirely towards NORX when no hardware support is available: here the AES-GCM versions only reach speeds of 49.30 and 56.30 cpb whereas NORX runs at 15.92 and 4.92 cpb, which corresponds to differences of factors 3.1 and 11.44, respectively.

On the Cortex-A8 chip from Table 21, we obtained speeds of 38.01 and 45.28 cpb for the optimised versions of AES-GCM and speeds of 10.50 and 8.96 cpb for the optimised versions of NORX which corresponds to speed-up-factors of 3.62 and 5.05. For the unoptimised variants we get 103.00 and 114.24 cpb for AES-GCM and 19.76 and 25.82 for NORX which are improvements by factors of 5.21 and 4.42.

In summary, it is very obvious that NORX is performance-wise more than on par with AES-GCM on the investigated platforms. The latter has only a slight advantage on platforms where the AES-NI extensions are available. In all other cases our cipher easily outperforms AES-GCM.

Hardware. A comprehensive evaluation of AES-GCM synthesized on an 65 nm CMOS ASIC is presented in [189]. For an 8-way parallel version of AES-GCM running at a frequency of 641 MHz, the authors report throughputs of around 82 Gbps. Unfortunately, we do not have any data points for NORX on the above architecture. Recall that the purely sequential version NORX64-4-1 achieves about 10 Gbps, when synthesized on a 180 nm UMC ASIC and being operated at a frequency of 125 MHz. We expect, however, that NORX achieves similar if not even better results compared to the above state-of-the-art AES-GCM implementation when realised on an equally modern architecture as the above 65 nm CMOS ASIC and exploiting higher frequencies and parallelization.

4.5 Design Rationale

In this chapter, we motivate the design choices made in NORX. We pursue a top-down approach, starting with the general layout and going into the details of the cipher's components in the later sections.

4.5.1 The Parallel MonkeyDuplex Construction

The layout of NORX is based on the monkeyDuplex construction [44, 48], but enhanced by the capability of parallel payload processing on multiple lanes (see Figures 32 and 33). The *parallel monkeyDuplex construction* is similar to the tree-hashing mode for sponge functions [45]. It allows NORX to take advantage of multi-core processors and enables high-throughput hardware implementations. Associated data can be authenticated as header and/or trailer data but only on a single lane. We considered it not worth the effort to enable processing of A and B in parallel, as they are usually rather short. The number of encryption lanes is controlled by the parallelism degree $0 \leq \mathsf{d} \leq 255$ which is

a fixed instance parameter. Hence, two instances with distinct d values cannot decrypt data from each other. Obviously, the same holds for differing w and r values.

To ensure that the payload blocks on parallel lanes are encrypted with distinct key streams, we use the branching phase to include an id into each of the parallel lanes. For NORX the id is a simple counter. Once the parallel payload processing is finished, the states are re-combined in the merging phase and NORX advances to the processing of the trailer (if present) or generation of the authentication tag.

There exists a formal proof of security for the parallel duplex construction, i.e. for all $0 \leq \mathsf{d} \leq 255$, as already mentioned in Section 4.3. In [141] it is shown that NORX achieves security levels of

$$\mathsf{min}\{2^{b/2}, 2^c, 2^k\}$$

for authenticity and confidentiality, where b, c, and k are state, capacity, and key sizes, assuming an ideal underlying permutation and a nonce-respecting adversary. This security bound would allow to increase the rate for each NORX variant by two words, i.e. by 64 bits in case of NORX32 and by 128 bits for NORX64. As a consequence, the performance would increase by approximately 16%. However, NORX is already very fast in soft- and hardware, see Section 4.4.3, and we therefore see no necessity in increasing the rate. Instead we consider the additional capacity as an enhanced security buffer.

4.5.2 The Functions F, G, and H

One of the main design goals for NORX was to create an ARX-related cipher which relies neither on S-boxes nor on integer additions to introduce non-linearity. For example, relinquishing S-boxes helps to avoid side-channel attacks based on timing leaks. Instead, we aimed to use exclusively more hardware-friendly bitwise logical operations like `NOT`, `AND`, `OR`, or `XOR` and bit-shifts, where combinations like `OR` or `AND` with `XOR` provide the required non-linearity. Instead of ARX which stands for integer addition, rotation, and XOR, we therefore use the term *LRX* to classify those types of primitives where integer addition is replaced by a combination of bitwise logical operations.

The Function H

Addition of two integers x and y can be written as

$$x + y = (x \oplus y) + ((x \wedge y) \ll 1)$$

according to [28, 164]. In NORX we started from the above representation but replaced integer addition with bitwise logical `XOR` which results in the non-linear operation H of the following form

$$\mathsf{H} : \mathbb{F}_2^{2n} \to \mathbb{F}_2^n, (x, y) \mapsto (x \oplus y) \oplus ((x \wedge y) \ll 1) \ .$$

The function H mimics integer addition of two bit strings x and y with a 1-bit carry propagation which provides a slight diffusion of bits.

Since NORX is designed to be a permutation-based cipher, the core function has to be bijective and therefore H has to be bijective, too. For the proof, we assume that one of the input arguments of H is fixed, a requirement which is given by default in the round function of NORX. To show the bijectivity of H, we show how to invert it. Therefore, let

$$z = (x \oplus y) \oplus ((x \wedge y) \ll 1)$$

with n-bit words x, y and z. We assume that y is fixed and write $x = \sum_{i=0}^{n-1} x_i \cdot 2^i$, $y = \sum_{i=0}^{n-1} y_i \cdot 2^i$, and $z = \sum_{i=0}^{n-1} z_i \cdot 2^i$ with x_i, y_i, and $z_i \in \mathbb{F}_2$. Writing down the inverse non-linear operation at bit level is then straightforward:

$$\begin{aligned}
x_0 &= (z_0 \oplus y_0) \\
x_1 &= (z_1 \oplus y_1) \oplus (x_0 \wedge y_0) \\
&\vdots \\
x_i &= (z_i \oplus y_i) \oplus (x_{i-1} \wedge y_{i-1}) \\
&\vdots \\
x_{n-1} &= (z_{n-1} \oplus y_{n-1}) \oplus (x_{n-2} \wedge y_{n-2}) \ .
\end{aligned}$$

This shows that H is indeed bijective under the above assumptions.

The Function G

The G function of NORX is inspired by the quarter-round function of the stream cipher ChaCha [40] which itself is an advancement of the quarter-round function of Salsa20 [107, 41] one of the eSTREAM finalists. Variants of ChaCha's quarter-round function can be found for example in the SHA-3 finalist BLAKE [222, 16] and its successor BLAKE2 [21].

Figure 40 shows how the G function of NORX transforms an input (a, b, c, d) compared to the quarter-round function of ChaCha. The rotation offsets for NORX are specified in Table 15. The offsets of ChaCha are $(s_0, s_1, s_2, s_3) = (16, 12, 8, 7)$ for 32-bit and $(s_0, s_1, s_2, s_3) = (32, 24, 16, 63)$ for 64-bit.[7]

The cyclic rotations are obviously bijective, H has been shown to be bijective too, and thus G is a permutation on the tuple (a, b, c, d). Further, it is a permutation when either of its input arguments is fixed, making it also a *latin square*.

[7] The original ChaCha stream cipher is defined for 32-bit words. For the 64-bit version we used the rotation offsets $(32, 24, 16, 63)$ from the BLAKE2 specification [21].

NORX			ChaCha		
a	$\longleftarrow$	$(a \oplus b) \oplus \big((a \wedge b) \ll 1\big)$	a	$\longleftarrow$	$a + b$
d	$\longleftarrow$	$(a \oplus d) \ggg r_0$	d	$\longleftarrow$	$(a \oplus d) \ggg s_0$
c	$\longleftarrow$	$(c \oplus d) \oplus \big((c \wedge d) \ll 1\big)$	c	$\longleftarrow$	$c + d$
b	$\longleftarrow$	$(b \oplus c) \ggg r_1$	b	$\longleftarrow$	$(b \oplus c) \ggg s_1$
a	$\longleftarrow$	$(a \oplus b) \oplus \big((a \wedge b) \ll 1\big)$	a	$\longleftarrow$	$a + b$
d	$\longleftarrow$	$(a \oplus d) \ggg r_2$	d	$\longleftarrow$	$(a \oplus d) \ggg s_2$
c	$\longleftarrow$	$(c \oplus d) \oplus \big((c \wedge d) \ll 1\big)$	c	$\longleftarrow$	$c + d$
b	$\longleftarrow$	$(b \oplus c) \ggg r_3$	b	$\longleftarrow$	$(b \oplus c) \ggg s_3$

Figure 40: Comparison of NORX and ChaCha core functions.

The Function F

The layout of the round function F of NORX is the same as used in ChaCha [40]. Recall that F transforms a state $s = (s_0, \ldots, s_{15})$ in two phases. First, F applies a column step col of the form

$$\mathsf{G}(s_0, s_4, s_8, s_{12}) \quad \mathsf{G}(s_1, s_5, s_9, s_{13}) \quad \mathsf{G}(s_2, s_6, s_{10}, s_{14}) \quad \mathsf{G}(s_3, s_7, s_{11}, s_{15})$$

and then a diagonal step diag of the shape

$$\mathsf{G}(s_0, s_5, s_{10}, s_{15}) \quad \mathsf{G}(s_1, s_6, s_{11}, s_{12}) \quad \mathsf{G}(s_2, s_7, s_8, s_{13}) \quad \mathsf{G}(s_3, s_4, s_9, s_{14})$$

see Algorithm 6 and Figure 34. Since G is a permutation, F is obviously a permutation, too. This means that there exist no states s and s', with $s \neq s'$, which produce the same result, i.e. $\mathsf{F}^{\mathsf{r}}(s) = \mathsf{F}^{\mathsf{r}}(s')$, after any number of rounds r. This characteristic of F is important for the duplex construction [44, 48] in order to retain some desirable security properties.

One great advantage of the ChaCha-related layout of F is that the modification of a single bit in the input has the chance of affecting all 16 output words after only one application of F which greatly enhances diffusion. Another benefit of the layout is the ability to execute the four applications of G during the steps col and diag completely in parallel which improves performance.

4.5.3 Selection of Constants

Rotation Offsets

The rotation offsets (r_0, r_1, r_2, r_3), see Table 15, used by NORX provide a good balance between security and efficiency. The concrete values r_i, with $0 \leq i \leq 3$, were selected according to the following conditions:

- At least two out of four offsets are multiples of 8.
- The remaining offsets are odd and have the form $8m \pm 1$ or $8m \pm 3$, with a preference for the first shape.

The motivation behind those criteria is the following: an offset which is a multiple of 8 preserves byte alignment and thus is much faster than an unaligned rotation on many non-64-bit architectures. Many 8-bit microcontrollers have only 1-bit shifts of bytes, so for example rotations by 5 bits are particularly expensive. Using aligned rotations, i.e. permutations of bytes, greatly increases the performance of the entire algorithm. Even 64-bit architectures benefit from such aligned rotations, for example when an instruction sequence of two shifts followed by `XOR` can be replaced by SSSE3's byte shuffling instruction `pshufb`. Odd offsets break up the byte structure and therefore increase diffusion.

In order to find good rotation offsets satisfying the above properties and assess their diffusion properties, we used an automated search combined with a diffusion test. The test analyses the diffusion behaviour of F^{r}, with $\mathsf{r} \geq 1$, parametrised with a given rotation tuple $r = (r_0, r_1, r_2, r_3)$, with $r_i \in \{1, \ldots, \mathsf{w} - 1\}$, on 1-bit input differences. We denote this parametrisation by $F^{\mathsf{r}}[r]$. By diffusion behaviour we mean here statistical values of the Hamming weight of the output difference of F^{r}. To be more precise, we looked for rotation offsets where the median of the Hamming weight of the output difference converges against $b/2$ for r as low as possible. In other words, the 0 and 1 entries in the output difference are equally distributed and exhibit no obvious structure. The test is described in detail in Algorithm 20.

Finally, we chose the offsets $(8, 19, 40, 63)$ for NORX64 and $(8, 11, 16, 31)$ for NORX32 which belonged to those offsets having very high values for average and median Hamming weight for $\mathsf{r} = 1$ and achieve full diffusion after $\mathsf{r} = 2$. Additionally, both offset tuples satisfy the initially specified conditions and offer good performance.

Table 22 lists the results of the test for 32- and 64-bit core functions with $1 \leq \mathsf{r} \leq 4$ and rotation offsets as specified above. The test results show that the diffusion speed of NORX's round function F is almost as high as ChaCha's and that full diffusion is reached after only two rounds. Figure 41 shows how single bit changes in the word s_0 propagate through the NORX state over the course of 5 steps ($= \mathsf{F}^{2.5}$). Unfortunately, there seems to be no combination of rotation values with three offsets being a multiple of 8 and one being $\mathsf{w} - 1$, like BLAKE2's $(32, 24, 16, 63)$, where F achieves a comparably strong diffusion as illustrated in Table 22. The reason for this can be traced back to the replacement of integer addition by the non-linear operation H of NORX.

Initialisation Constants

The four basic constants $u_0, \ldots, u_3$ of 32-bit and 64-bit NORX correspond to the first digits of π. The other six constants are derived iteratively from $u_0, \ldots, u_3$ by

$$(u_{4j+4}, u_{4j+5}, u_{4j+6}, u_{4j+7}) = \mathsf{G}(u_{4j}, u_{4j+1}, u_{4j+2}, u_{4j+3})$$

Algorithm 20: diffusion_test(r, R)

Inputs:
$\mathsf{r} \in \mathbb{N}$, $R \in \mathbb{Z}_{\mathsf{w}}^{4n}$
Output:
$L \in (\mathbb{Z}_{\mathsf{w}}^{4} \times \mathbb{N} \times \mathbb{N} \times \mathbb{Q} \times \mathbb{Q})^{m}$
Algorithm:

```
1.  L ← ∅
2.  for r ∈ R do
3.      L_r ← ∅
4.      for 1 to 10^6 do
5.          X ←$ {0, 1, 2, 3, 4, ..., 2^(b−1)}
6.          Y ←$ {1, 2, 4, 8, 16, ..., 2^(b−1)}
7.          s ← to_int_w(X)
8.          s' ← to_int_w(X ⊕ Y)
9.          x ← F^r[r](s) ⊕ F^r[r](s')
10.         L_r ← L_r ‖ hw(x)
11.     end
12.     L ← L ‖ (r, min(L_r), max(L_r), avg(L_r), median(L_r))
13. end
14. return L
```

for $j \in \{0, 1\}$. The complete list of constants is depicted in Table 23. The main purpose of the constants is to bring asymmetry during initialisation and to limit the freedom of an attacker where he might inject differences.

Domain Separation Constants

The NORX algorithm is separated into different data processing phases. Each phase uses its own domain separation constant to mark certain events like the absorbing of data blocks or merging and branching steps in case of an instance with parallelism degree $\mathsf{d} \neq 1$. A domain separation constant is always added to the least significant byte of the capacity word s_{15}. The constants are given in Table 16. The separation of the processing phase is important for the security proofs of the indifferentiability of the (parallel) duplex construction [46, 48, 141]. In addition, they help to break the self-similarity of the round function and thus increase the complexity of certain kind of attacks on NORX, for example, like slide attacks, see Section 5.2.4.

4.5.4 Number of Rounds

For a higher protection of the key and authentication tag, e.g. against differential or algebraic cryptanalysis, we chose twice the number of rounds for initialisation and finalisation, i.e. $\mathsf{F}^{2\mathsf{r}}$, compared to the data processing phases which use F^{r}. This method was already proposed in [44] and has only minor effects on the overall performance, but

Table 22: Diffusion statistics for (inverse) round functions of NORX and ChaCha.

	NORX32				ChaCha (32-bit)				Inverse NORX32				Inverse ChaCha (32-bit)			
r	min	max	avg	median	min	max	avg	median	min	max	avg	median	min	max	avg	median
1	83	280	179.222	181	73	294	182.195	185	17	162	49.444	47	17	126	44.776	44
2	194	307	256.024	256	199	312	255.999	256	160	306	247.737	248	164	304	244.982	246
3	198	312	255.995	256	204	313	255.988	256	202	307	255.991	256	203	310	255.994	256
4	201	307	255.996	256	200	314	255.989	256	202	315	256.018	256	200	311	256.022	256
	NORX64				ChaCha (64-bit)				Inverse NORX64				Inverse ChaCha (64-bit)			
r	min	max	avg	median	min	max	avg	median	min	max	avg	median	min	max	avg	median
1	95	429	230.136	222	73	506	248.843	246	17	203	51.346	49	17	142	46.129	45
2	440	589	511.982	512	430	591	512.013	512	262	568	433.742	435	194	543	382.667	383
3	434	589	512.008	512	439	589	511.971	512	440	593	511.995	512	440	591	511.964	512
4	428	589	511.986	512	435	585	512.008	512	435	585	512.011	512	433	596	511.991	512

Table 23: Initialisation constants of NORX.

	NORX32	NORX64		NORX32	NORX64
u_0	243F6A88	243F6A8885A308D3	u_5	38531D48	670A134EE52D7FA6
u_1	85A308D3	13198A2E03707344	u_6	839C6E83	C4316D80CD967541
u_2	13198A2E	A4093822299F31D0	u_7	F97A3AE5	D21DFBF8B630B762
u_3	03707344	082EFA98EC4E6C89	u_8	8C91D88C	375A18D261E7F892
u_4	254F537A	AE8858DC339325A1	u_9	11EAFB59	343D1F187D92285B

should increase the security margin of NORX against non-generic attacks. The minimal value of $\mathsf{r} = 4$ is based on the following observations:

- The diffusion experiments, as presented in Section 4.5.3, show that $\mathsf{r} = 2$ is required to get full diffusion.
- The best attacks on Salsa20 and ChaCha [14, 225, 233] theoretically break 8 and 7 rounds, respectively, which roughly corresponds to 4 and 3.5 rounds of the NORX core permutation. However, those attacks are based on a much stronger attack model which cannot be used for the duplex construction of NORX.
- The cryptanalysis of NORX as presented in Section 5. The best differentials we were able to find in the permutation F^{r} belong to a class of high-probability truncated differentials over 1.5 rounds, see Section 5.3.1, and a class of impossible differentials over 3.5 rounds, see Section 5.3.2. These are not applicable to NORX though.

The number of rounds may be adjusted according to the future cryptanalytic results on NORX.

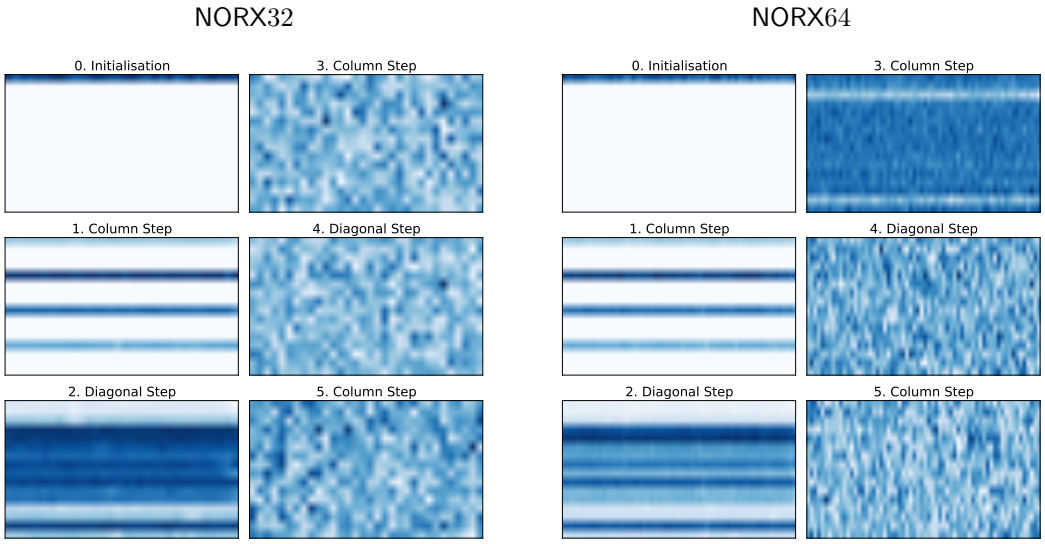


Figure 41: Visualisation of NORX diffusion.

4.5.5 The Padding Rule

The sponge (or duplex) construction offers protection against generic attacks if the padding rule is sponge-compliant, i.e. if it is injective and ensures that the last block is different from the all-zero block. In [45] it has been proven that the multi-rate padding satisfies those properties. Moreover, it is simple to describe, easy to implement, and very efficient and was thus a natural choice to be used in NORX. Additionally, the multi-rate padding increases the complexity to mount certain kind of attacks on NORX, like slide attacks, see Section 5.2.4.

4.6 Conclusion

This chapter gave a comprehensive introduction to NORX, a new, state-of-the-art authenticated encryption scheme with support for associated data. The cipher is a candidate of the CAESAR competition which started in early 2014. Next to its specification, we discussed the cipher's most important features, presented results from our extensive software performances measurements on multiple platforms as well as those from an (external) hardware evaluation of an ASIC design, and gave insights into the design process and the motivation behind our design decisions.

There are several options for future research projects targeting NORX: on the software side, one interesting topic is the evaluation of NORX on more resource-constrained platforms, like microcontrollers, and how the scheme compares to currently deployed

solutions and to the other CAESAR candidates. On the hardware side, a similar evaluation of NORX implementations for FPGAs would be of very high interest as well.

Furthermore, we are currently working on versions of NORX based on 8- and 16-bit words, which highly reduce the state sizes and could be of special interest for environments having access to only very limited resources.

Another important topic is of course the security analysis of the scheme. We present first results in Chapter 5. However, an extensive security analysis of any new cipher requires a lot of effort from many researchers. We thus invite and encourage the readers to analyse the security of NORX.

Finally, regarding intellectual property, we explicitly state that NORX is available on a royalty-free basis, the algorithm is not covered by a patent, and the designers of NORX do not have any plans to file a patent application in the future.

Chapter 5

Analysis of NORX

5.1 Introduction

This chapter is dedicated to the security analysis of NORX. We focus on differential and rotational aspects of the round permutation F^r and use our results to draw conclusions on the security of the complete scheme.

The analysis of NORX was presented at Latincrypt 2014 [19].

Outline. Section 5.2 discusses general observations on the core functions G and F, and the implications that follow for the security of NORX. Section 5.3 is dedicated to the differential cryptanalysis of NORX. First, we introduce the required notation, then we describe how to construct simple differentials by analysing 1-bit input differences to the G permutation. Finally, we introduce NODE, the NORX *Differential Search Engine* which allows to search for differential trails in F^r, and present the results from our extensive experiments. In Section 5.4, we analyse rotational aspects of F^r, and Section 5.5 concludes the chapter.

5.2 General Observations on G and F

5.2.1 Fix Points

A *fix point* x of a permutation P is an input that satisfies

$$P(x) = x \ .$$

This property also extends to the n-fold iteration of P, i.e. a fix point x of P is also a fix point of P^n.

In our first step of the analysis of NORX, we are interested in whether there exist any fix points in the 32- and/or 64-bit permutation F^r. To this end we start with the examination of G, the basic building block of F and F^r. Recalling the layout of G (see Figure 6), we almost immediately see that

$$\mathsf{G}(0,0,0,0) = (0,0,0,0)$$

holds. In other words, the all-zero input is a fix point of G. Moreover, since col and diag are just four-fold parallel applications of G (see Figure 34), it follows trivially that the all-zero input also results in a fix point for those two functions and obviously for F and F^r as well.

An attacker could use the zero-to-zero point as a trivial distinguisher to attack NORX. However, it seems hard to exploit this property for an actual attack, as hitting the all-zero state is as hard as hitting any other arbitrary state. Thus, the ability to hit a predefined state implies the ability to recover the key which is equivalent to completely breaking NORX. Therefore, the zero-to-zero point is not a significant threat to the security of NORX.

The more interesting question is, whether there are any other fix points in the NORX permutations besides the trivial one, and if so, how many. To investigate this question, we formulated the problem for the 32- and 64-bit variants of G in CVC which is the input language of the SMT solver STP [116]. For the 32-bit version, the solver was indeed able to verify that there are no further fix points which implies that the same holds for the 32-bit round function F. However, for the 64-bit version of G, the problem has a much higher complexity: even after over 1000 hours, STP was unable to finish its computation with a positive or negative result. Although we were not able to answer the fix point question for this case using the above approach, we consider it to be very unlikely that there are further fix points besides the trivial one. Note that even if there are further fix points, NORX implements a couple of countermeasures to prevent an attacker from exploiting them: first, the initial state is loaded with asymmetric constants, which should make it very hard for an attacker to construct a fix point, even in the worst-case scenario where he can control nonce and key words. Second, as soon as initialisation is finished, an attacker can influence only the ten rate words, while the six capacity words are uncontrollable and unknown to him, which likely makes it infeasible to attack NORX through fix points in this scenario either. The third countermeasure are the domain separation constants which are integrated into the state before each application of F^r and provide another source of asymmetry.

5.2.2 Weak States

The category of states which exhibit a non-pseudo-random behaviour when transformed with F^r, are called *weak states*. With fix points we already saw one type of states that belongs to this category. However, the more general form of weak states for NORX are of the following shape

$$\begin{pmatrix} w & w & w & w \\ x & x & x & x \\ y & y & y & y \\ z & z & z & z \end{pmatrix}$$

with w, x, y, and z being arbitrary w-bit sized words. The column-pattern is preserved by F^{r} for any value of $\mathsf{r} > 0$. These weak states hold obviously for both col and diag, i.e. it does not matter if the state is transformed first by col and then by diag or the other way around. The ability to hit such a state purposely, is equivalent to the ability of reconstructing the key and therefore breaking the entire scheme. Although there are quite many of these states, namely $2^{4\mathsf{w}}$, their number is still negligible compared to the total number of $2^{16\mathsf{w}}$ states. Thus, the probability to hit such a state is $2^{-12\mathsf{w}}$ which translates to probabilities of 2^{-384} ($\mathsf{w} = 32$) and 2^{-768} ($\mathsf{w} = 64$). In other words, an attacker is better off brute-forcing the key, which has success probabilities of 2^{-128} ($\mathsf{w} = 32$) and 2^{-256} ($\mathsf{w} = 64$), respectively, than attacking NORX through weak states. Additionally, as for the fix point scenario, duplex construction, asymmetric initialisation, and domain separation constants provide an extra level of protection against weak state attacks. In conclusion, weak states should pose a very small threat to the security of NORX.

5.2.3 Algebraic Properties

Algebraic attacks on cryptographic algorithms, as discussed in the literature [8, 13, 89, 105], target ciphers whose internal state is mainly updated in a linear way and thus exploit a low algebraic degree of the attacked primitive. However, this is not the case for NORX, where the $b \in \{512, 1024\}$ inner state bits are updated in a strongly non-linear fashion. In the following, we briefly discuss the algebraic properties of NORX and demonstrate why it is unlikely that algebraic attacks can be successfully mounted against the cipher.

A convenient way to represent a Boolean function is through its *Algebraic Normal Form* (ANF). Given a Boolean function $f : \mathbb{F}_2^n \longrightarrow \mathbb{F}_2$, the ANF of f can be modelled as a multivariate polynomial, i.e. a sum of monomials in n input variables. Both a large number of monomials in the ANF and a good distribution of their degrees are important properties of non-linear building blocks in ciphers [24].

We constructed the ANF of G and measured the degree of every of the 4w polynomials and the distribution of the monomials. Table 24 reports the number of polynomials per degree as well as information on the distribution of monomials for both versions of G.

Table 24: Algebraic properties of G.

#polynomials by degree						
w	3	4	5	6	7	8
32	2	6	58	2	8	52
64	2	6	122	2	8	116

#monomials				
w	min	max	avg	median
32	12	489	242	49.5
64	12	489	253	49.5

In both cases most polynomials have degree 5 or 8 and merely 2 have degree 3. Multiplying each of the above values by 4 gives the distribution of degrees for the whole

state after a single col or diag step. Due to memory constraints, we were unable to construct[1] the ANF for a single full round F, neither for the 64-bit nor for the 32-bit version. In summary, we see that the state of NORX is updated in a strongly non-linear fashion. Due to the rapid growth of the degree, a large number of monomials and state size of 512 and 1024 bits, we believe that it is unlikely that algebraic cryptanalysis can be used to successfully mount an attack on the AEAD scheme.

5.2.4 Slide Attacks

Slide attacks try to exploit the symmetry in a primitive that consists of the iteration of a number of identical rounds. They were introduced by Biryukov et al. [65, 66] for the cryptanalysis of block ciphers. Later, they were also extended to stream ciphers [208] and hash functions [119]. To protect sponge constructions against slide attacks two simple countermeasures can be found in the literature:

1. The first protection, as introduced in [119], can be implemented by adding a non-zero constant to the state just before the permutation is applied.

2. The second, as proposed in [205], recommends to use a message padding, which ensures that the last processed data block is different from the all-zero message.

These countermeasures should also hold for the duplex construction, since it also belongs to the sponge function family and thus for NORX. Both defensive mechanisms are already integrated into the cipher: the domain separation constants are added to the state just before the permutation F^r is applied and the multi-rate padding ensures that the last processed data block is different from the all-zero block. See Section 4.2.5 for more information on padding and domain separation. All in all, slide attacks should pose little to no threat to NORX.

5.3 Differential Cryptanalysis

It is crucial to analyse the resistance of any new designs against differential attacks. In this section we therefore present our results on the differential cryptanalysis of NORX.

The basic notions of differential cryptanalysis have been introduced in Section 1.2.2. First, we analyse the propagation of differences with 1 active bit through the G function, and derive from that a few high-probability truncated differentials of low weight for a few steps of the core permutation F^r. Afterwards, we also study impossible differential cryptanalysis.

The search for differentials by hand becomes infeasible very quickly and especially for a larger number of rounds. The second part of this section is dedicated therefore to more

[1] Using SAGE [229] on a workstation with 64 GiB RAM.

advanced differential search methods in F^r. We discuss an approach how to embed the differential search in a more general framework which allows to automate the task by exploiting the power of SAT- and SMT-solvers.

5.3.1 Simple Differentials

In this section, we analyse differential propagation in the permutations G and F of NORX, show how to compute some simple differentials together with their corresponding weights. To be more precise, if we speak in the following of differentials or (differential) characteristics, we always mean XOR-differentials or XOR-characteristics.

Simple Differentials in G

First, we start with the simplest case, where an input difference of G has only one active bit. Since the direct analysis of G is already rather complex we decompose G into two functions G_1 and G_2 and analyse initially only the behaviour of G_1. Therefore, let $\mathsf{G}_1 : \mathbb{F}_2^{4\mathsf{w}} \longrightarrow \mathbb{F}_2^{4\mathsf{w}}$ be a vector Boolean function defined as

$$\begin{aligned} a &\longleftarrow (a \oplus b) \oplus ((a \wedge b) \ll 1) \\ d &\longleftarrow (a \oplus d) \ggg r_0 \\ c &\longleftarrow (c \oplus d) \oplus ((c \wedge d) \ll 1) \\ b &\longleftarrow (b \oplus c) \ggg r_1 \end{aligned}$$

and let G_2 be specified analogously but with rotation offsets r_2 and r_3 instead of r_0 and r_1. Thus, it obviously holds that

$$\mathsf{G}(a, b, c, d) = \mathsf{G}_2(\mathsf{G}_1(a, b, c, d))$$

for all $(a, b, c, d) \in \mathbb{F}_2^{4\mathsf{w}}$. Now let $\alpha, \beta \in \mathbb{F}_2^{4\mathsf{w}}$ be input and output differences of a differential in G_1 with $\alpha = (\alpha_0, \alpha_1, \alpha_2, \alpha_3)$ and $\beta = (\beta_0, \beta_1, \beta_2, \beta_3)$, i.e. we have

$$\alpha \xrightarrow{\mathsf{G}_1} \beta \ .$$

Further, let $v \in \{0, \ldots, 3\}$ and assume that $\mathsf{hw}(\alpha_v) = 1$ with an active bit at position $i \in \{0, \ldots, \mathsf{w} - 1\}$. Moreover, assume that $\mathsf{hw}(\alpha_u) = 0$ for all $u \in \{0, \ldots, 3\} \setminus \{v\}$. For the computation of the active-bit indices, we denote by $s(i)$ and $r(i)$ the index-level equivalent operations to the shift $\ll 1$ and cyclic rotation $\ggg r$ of the ith bit. These operations translate to $s(i) = i + 1$ and $r(i) = (i - r) \bmod \mathsf{w}$. Moreover, each application of s increases the weight w of the active bit by 1, since it is only applied in the non-linear operation H of NORX. The active bits in the output difference β and associated differential weights w after G_1 are depicted in Table 25.

Table 25: Active bits in $\beta = \mathsf{G}_1(x) \oplus \mathsf{G}_1(x \oplus \alpha)$ starting from an active bit index i in α.

	α_0	w	α_1	w	α_2	w	α_3	w
β_0	$j_{0,0} = i$ $j_{0,1} = s(i)$	0 1	$j_{0,2} = i$ $j_{0,3} = s(i)$	0 1	–		–	
β_1	$j_{1,0} = r_1 \circ r_0(i)$ $j_{1,1} = r_1 \circ s \circ r_0(i)$ $j_{1,2} = r_1 \circ r_0 \circ s(i)$ $j_{1,3} = r_1 \circ s \circ r_0 \circ s(i)$	0 1 1 2	$j_{1,4} = r_1 \circ r_0(i)$ $j_{1,5} = r_1 \circ s \circ r_0(i)$ $j_{1,6} = r_1(i)$ $j_{1,7} = r_1 \circ r_0 \circ s(i)$ $j_{1,8} = r_1 \circ s \circ r_0 \circ s(i)$	0 1 0 1 2	$j_{1,9} = r_1(i)$ $j_{1,10} = r_1 \circ s(i)$	0 1	$j_{1,11} = r_1 \circ r_0(i)$ $j_{1,12} = r_1 \circ r_0 \circ s(i)$	0 1
β_2	$j_{2,0} = r_0(i)$ $j_{2,1} = s \circ r_0(i)$ $j_{2,2} = r_0 \circ s(i)$ $j_{2,3} = s \circ r_0 \circ s(i)$	0 1 1 2	$j_{2,4} = r_0(i)$ $j_{2,5} = s \circ r_0(i)$ $j_{2,6} = r_0 \circ s(i)$ $j_{2,7} = s \circ r_0 \circ s(i)$	0 1 1 2	$j_{2,8} = i$ $j_{2,9} = s(i)$	0 1	$j_{2,10} = r_0(i)$ $j_{2,11} = r_0 \circ s(i)$	0 1
β_3	$j_{3,0} = r_0(i)$ $j_{3,1} = r_0 \circ s(i)$	0 1	$j_{3,2} = r_0(i)$ $j_{3,3} = r_0 \circ s(i)$	0 1	–		$j_{3,4} = r_0(i)$	0

The first insight is that in some cases different "paths" lead to the same bit index, i.e. active bits could overlap and cancel each other out as a consequence. This occurs for $j_{1,1}$ and $j_{1,2}$, $j_{2,1}$ and $j_{2,2}$, $j_{1,5}$ and $j_{1,7}$, and $j_{2,5}$ and $j_{2,6}$. In these cases, the total probability that the bit is active is

$$2^{-w} + 2^{-w} - 2^{-w} \cdot 2^{-w} = \frac{2^{w+1} - 1}{2^{2w}}$$

resulting in a probability of $\frac{3}{4}$ for $w = 1$. Considering these overlaps, the total number of active bits in words α_0, α_1, α_2, and α_3, respectively, is at most 10, 11, 4, and 5.

Another event that could neutralise active bits is the shift operation in H. This happens if the active bit is at index $\mathsf{w} - 1$ before the application of H. The shift then "moves" the active bit beyond the word boundary thereby erasing it. As a consequence, all bits depending on the erased index can not become active. For example, if bit $i = \mathsf{w} - 1$ is active in α_0, then the difference $j_{0,1} = s(i)$ is erased by the shift operation, which has the effect that bits $j_{3,1}$, $j_{2,2}$, $j_{1,2}$, $j_{2,3}$, and $j_{1,3}$ remain inactive. These bits can become active through the propagation of other active bits though. Figure 42 visualises the dependencies between the active bits from Table 25. The nodes of the tree are the active bit indices at the output of G_1. The labels (f, v) on the edges show the operation $f \in \{id, s, r_0, r_1\}$ that leads from one active bit to the next and the corresponding weight increase $v \in \{0, 1\}$. The differential weight w of an active bit at the output of G_1 can be computed by summing over all the values v on the edges of a path leading from the root of the tree to a particular node. This results in the same weights as already shown in Table 25.

The differentials in Table 25 only hold for input differences having exactly one active bit. If we consider input differences with more than one active bit, the situation gets immediately a lot more complex, because active bits interact with each other more often and the effects become hard to predict in general. For example, an input difference having active bits $\mathsf{w} - 1$ in both α_0 and α_1 leads to a cancellation of the active bit $j_{0,0}$

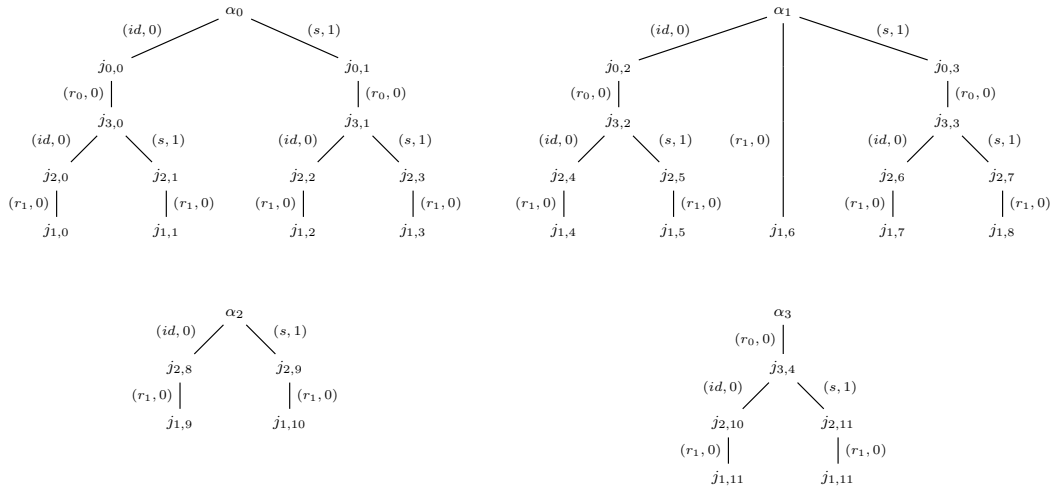


Figure 42: Relations between active bits in G_1.

of probability 1 during the update of word a in G_1. We show below how this property can be exploited to build some simple differentials for G having high probability and low weight output differences.

Now that we know how active bits propagate in G_1, we want to use that knowledge to describe active bits at the output of G. Differential trails in G can obviously be described as follows:

$$\alpha \xrightarrow{\mathsf{G}_1} \beta \xrightarrow{\mathsf{G}_2} \gamma \ .$$

Hence, the active output bits in β of G_1 are the new active input bits of G_2. These then propagate to the output γ of the function G_2 which are equivalent to the output bits of G. Under the assumption that no cancellation of active bits occurs and using our previous computations on the number of active bits at the output of G_1, we can compute upper bounds on the active bits after one application of G quite easily. Recall that for a single active bit in α_0 at the input of G_1 we get at most 10 active bits at the output. To be more precise, we get 2 active bits in β_0, 3 in β_1, 3 in β_2, and 2 in β_3, see the first column in Table 25. To compute the active output bits in G_2, we can obviously re-use the results from G_1, since both functions only differ in the rotation offsets. Thus, using Table 25 on the above active bits in β, we see that we get at most

$$2 \cdot 10 + 3 \cdot 11 + 3 \cdot 4 + 2 \cdot 5 = 75$$

active bits at the output γ of G_2 or equivalently of G. Repeating the above calculations for single active bits in the remaining three input words α_1, α_2, and α_3, we get upper bounds of 86, 30, and 35 for the number of active bits. These bounds are not tight, though. Analysing the formulas from $\mathsf{G}_2(\mathsf{G}_1(a, b, c, d))$ in detail, one realises that quite

a few active bits overlap, and that there are often multiple paths that lead to the same active bit. The tight upper bounds of active bits are listed in the Table 26.

Table 26: Maximum number of active output bits in G on a single active input bit.

single active bit in	α_0	α_1	α_2	α_3
max. active bits in γ	51	62	22	27

We also verified these numbers experimentally for both the 32- and 64-bit version of G, i.e. we found differential trails that have the predicted number of active output bits on a single active input bit.

Simple Differentials in F^r

Now that we have analysed how a single active bit propagates through G we want to use this knowledge to find some simple differentials in F of NORX. We analyse again both variants of G.

We first consider a simple attack model where the initial state is assumed to be chosen uniformly at random and where one seeks differences in the initial state that give biased differences in the state obtained after a small number of iterations of F. High-probability truncated differentials wherein the output difference concerns only a small subset of bits (e.g., a single bit) are sufficient to distinguish a (reduced-round) permutation from a random one and are easier to find for an adversary than differentials on all b bits of the state. To find such differentials we start from our previous analysis of G and extend it to F^r. We observe that it is easy to track differences during the first few steps and in particular to find probability-1 (truncated) differential characteristics for a small number of iterations of F. We found three notable differentials in G, see Table 27, that have high probability and an output difference of low Hamming weight.

Table 27: High-probability differentials of low weight for 32- and 64-bit G.

	Differences				w
α	80000000	80000000	80000000	00000000	0
β	00000000	00000001	80000000	00000000	
α	00000000	80000000	80000000	80000000	1
β	80000000	00010001	80008000	00008000	
α	00000000	80000000	80000000	80000000	1
β	80000000	00030001	80018000	00008000	

	Differences				w
α	8000000000000000	8000000000000000	8000000000000000	0000000000000000	0
β	0000000000000000	0000000000000001	8000000000000000	0000000000000000	
α	0000000000000000	8000000000000000	8000000000000000	8000000000000000	1
β	8000000000000000	0000000001000001	8000000000800000	0000000000800000	
α	0000000000000000	8000000000000000	8000000000000000	8000000000000000	1
β	8000000000000000	0000000003000001	8000000001800000	0000000000800000	

To prove that the first differentials of their respective word sizes in Table 27, indeed have probability 1 or equvialently weight 0, we look again at the decomposition $\mathsf{G} = \mathsf{G}_2 \circ \mathsf{G}_1$

and construct the differential

$$\alpha \xrightarrow{\mathsf{G}_1} \beta \xrightarrow{\mathsf{G}_2} \gamma$$

in two steps. First, consider an active bit at index $i = \mathsf{w} - 1$, in α_0. By looking at the first tree in Figure 42, we see that at most 6 out of 10 bits can become active, namely bits $j_{0,0}$, $j_{3,0}$, $j_{2,0}$, $j_{2,1}$, $j_{1,0}$, and $j_{1,1}$ which are located in the left subtree. The bits in the right subtree remain inactive, since $s(i)$ is erased due to the shift $\ll 1$. Similar considerations hold for α_1, where at most 7 out of 11 bits can become active. For α_2 and α_3 there will be exactly 2 and 3 active bits at the output of G_1. The probability-1 differential has active bits at indices $i_0 = i_1 = i_2 = \mathsf{w} - 1$ in input differences α_0, α_1, and α_3, respectively. The output difference β_0 of G_1 has no active bit since $s(i_0)$ and $s(i_1)$ are erased, and i_0 and i_1 cancel each other out during the update of word a. As a consequence, also β_3 has no active output bit. In β_2 only bit i_2 is active since it propagates directly from the input. Finally, β_1 has no active output bits since i_1 and i_2 cancel each other out during the update of word b. In summary, only bit $i_2 = \mathsf{w} - 1$ is active in the word β_2 at the output G_1. Referring again to Table 25, we easily see that bits $\mathsf{w} - 1$ and $r_3(\mathsf{w} - 1) = 0$ are active with probability 1 in the words γ_2 and γ_3 at the output of G_2 and G, respectively.

Similar considerations show that the 2nd and 3rd differential from Table 27 have weight 1 or equivalently probability 1/2. We do not go into the details at this point, though.

Applying those differentials to F has the effect that the diffusion of the state is delayed by one step. We also examined differentials with input differences having other combinations of active MSBs, which lead to similar output differences, but none having a lower or equal Hamming weight compared to those from Table 27. Using the first of the above differentials, we were able to easily derive a truncated differential over 3 steps (i.e. $\mathsf{F}^{1.5}$) which has probability 1. This truncated differential can be used to construct an impossible differential over 3.5 rounds for the 64-bit version of F which is shown in Section 5.3.2 below.

We expect that advanced search techniques should be able to find better differential distinguishers for a higher number of iterations of F, such that the sparse difference occurs at a later step than the first. Nevertheless, we expect that it is not possible to find differential distinguishers for as many rounds as specified for our instances, see Table 14, taking into account the reduced freedom an adversary has when attacking the initialisation or round permutation.

5.3.2 Impossible Differentials

Cryptanalysis using impossible differentials was introduced in 1998 by L. Knudsen to attack the block cipher DEAL [160]. Later it was extended by E. Biham et al. in order to attack the block ciphers Skipjack [54] and IDEA [55]. The latter introduces the so-called *miss-in-the-middle* technique. This approach combines two probability-1 differentials, one in forward and one in backward direction which exhibit a conflict when both directions

are joined. This strategy leads to an impossible event, i.e. an incident having probability 0, and can be used to construct distinguishers or even mount key recovery attacks.

In our case, we construct an impossible differential over 3.5 rounds of the 64-bit version of F, namely 3 steps in forward and 4 steps in backward direction, using the miss-in-the-middle approach from above. The impossible differential is thus of the form

$$\alpha \xrightarrow{\mathsf{F}^{1.5}} \beta \neq \beta' \xleftarrow{\mathsf{F}^{-1.5} \circ \mathsf{col}^{-1}} \alpha'$$

with differences α, β, α', and β' as given in Table 28. The symbols * and ? denote partially known and unknown entries, respectively. The conflict occurs in the 2nd bit of the 14th word of the differences β and β'. In forward direction, this bit has always value 1 whereas in backward direction it has always value 0. The conflicting nibbles are marked in red in Table 28 and are of the form $*100_2$ and 0000_2 for β and β', respectively. We validated the impossible differential empirically in about 2^{32} runs.

Note that there are many more impossible differentials of the above type starting from comparable input differences in forward and backward direction. Nevertheless, using such a simple approach, we were not able to construct impossible differentials stretching over more than 3.5 rounds.

Table 28: Impossible differential over 3.5 rounds of 64-bit F.

α				β			
8000000000000000	0000000000000000	0000000000000000	0000000000000000	???*?*???***?**?	*0**??*000000***	***0?*?*00**000*	00**??*00***?***
8000000000000000	0000000000000000	0000000000000000	0000000000000000	???*????********	?***??*????*?*??	??*0???*0?*?*??*	??**?*?????*?*??
8000000000000000	0000000000000000	0000000000000000	0000000000000000	????*????*??????	?*00000????*?*??	??*0*?**0**?*??*	??**?*????****??
0000000000000000	0000000000000000	0000000000000000	0000000000000000	???***??*????*?*	*000000****0**??	?*00**0*0*****0**	******?**?*****??
α'				β'			
0000000000000000	0000000000000000	8000000000000000	0000000000000000	????????????????	????????????????	????????????????	????????????????
0000000000000000	0000000000000000	0000000000000000	0000000000000000	????????????????	????????????????	????????????????	????????????????
0000000000000000	0000000000000000	0000000000000000	0000000000000000	?????????????????	??????????????80	?????????????????	?????????????????
0000000000000000	0000000000000000	0000000000000000	0000000000000000	????????????????	????????????????	????????????????80	????????????????

While it was relatively easy to construct the above impossible differential, it cannot be used to attack (round-reduced) NORX, due to the following reasons:

- The state setup used during initialisation prevents an attacker from setting the required input difference in forward direction. It would be necessary to set differences in the first three consecutive MSBs of a column, which is impossible, as every column is initialised with at least two constant values (see Figure 9). Thus, even in a scenario where an attacker can influence key and nonce during initialisation, it is not possible to exploit this class of impossible differentials.
- During data processing, where the proper differences could be set theoretically in the rate words of the state, it is not possible to use that impossible differential

either: Under the assumption that an attacker is nonce-respecting [213], which is a basic requirement for the operation of NORX, and that F^{r} provides maximum security for $\mathsf{r} \geq 4$, two states being set up with two different nonces lead to two distinct internal states after the initialisation phase. Therefore, an attacker does not know how to set header blocks to construct the required input difference in forward direction, since he is not capable of influencing the (unknown) capacity words of the state. The same holds for the payload phase. Thus, the impossible differential cannot be exploited in a later phase of the algorithm either.

5.3.3 NODE – NORX Differential Search Engine

This section is dedicated to the automation of differential cryptanalysis of NORX. First, we introduce the mathematical models required to describe differential propagation in F^{r} of NORX. Then we describe NODE, a differential search framework for the NORX permutation and finally present our results.

Mathematical Models

Let $n \in \mathbb{N} \setminus \{0\}$ denote the word size. For the analysis of NORX, we have $n = \mathsf{w} \in \{32, 64\}$. Let x and y denote bit strings of size n and let α, β, and γ denote differences of size n. In the following, we identify with α_i, β_i, γ_i, x_i, and y_i the individual bits of α, β, γ, x, and y, where $0 \leq i \leq n-1$. Recall that the non-linear operation H of NORX is a vector Boolean function of the form

$$\mathsf{H} : \mathbb{F}_2^{2n} \longrightarrow \mathbb{F}_2^n,\ (x, y) \mapsto (x \oplus y) \oplus ((x \wedge y) \ll 1)\ .$$

Referring to the notions introduced in Section 1.2.2, we see that an XOR-differential $(\alpha, \beta) \longrightarrow \gamma$ of H fulfils

$$\alpha \oplus \beta \oplus \gamma = ((x \wedge \beta) \oplus (y \wedge \alpha) \oplus (\alpha \wedge \beta)) \ll 1 \tag{5.1}$$

for n-bit strings x and y and n-bit differences α, β, and γ. Rewriting Equation 5.1 on bit level we get

$$\begin{aligned} 0 &= \alpha_0 \oplus \beta_0 \oplus \gamma_0 \\ 0 &= (\alpha_i \oplus \beta_i \oplus \gamma_i) \oplus (\alpha_{i-1} \wedge \beta_{i-1}) \oplus (x_{i-1} \wedge \beta_{i-1}) \oplus (y_{i-1} \wedge \alpha_{i-1}), \quad i > 0\ . \end{aligned}$$

Proposition 3 is an important step towards expressing differential propagation in NORX and is the analogue to Theorem 1 for integer addition from [178]. The proposition eliminates the dependence of Equation 5.1 on the bit strings x and y and therefore allows to check in a constant amount of word operations if a given tuple (α, β, γ) is an (impossible) XOR-differential of H. This is a very valuable result and, as we will see later, paves the way for automated differential cryptanalysis.

Proposition 3. *An* XOR*-differential* $(\alpha, \beta) \longrightarrow \gamma$ *of the non-linear operation* H *of* NORX *satisfies the following equation:*

$$(\alpha \oplus \beta \oplus \gamma) \wedge (\neg((\alpha \vee \beta) \ll 1)) = 0 \ . \tag{5.2}$$

Proof. On bit level Equation 5.2 has the form

$$\begin{aligned} 0 &= \alpha_0 \oplus \beta_0 \oplus \gamma_0 \\ 0 &= (\alpha_i \oplus \beta_i \oplus \gamma_i) \wedge (\alpha_{i-1} \oplus 1) \wedge (\beta_{i-1} \oplus 1), \quad i > 0 \ . \end{aligned}$$

Obviously, the least significant bits (i.e. $i = 0$) are identical for Equations 5.1 and 5.2. For $i > 0$, let $t = (\alpha_i \oplus \beta_i \oplus \gamma_i) \oplus (\alpha_{i-1} \wedge \beta_{i-1})$. If $t = 0$, then Equation 5.1 has always the solution $x_{i-1} = y_{i-1} = 0$. Otherwise, if $t = 1$, Equation 5.1 is only solvable if $\alpha_{i-1} = 1$ or $\beta_{i-1} = 1$. But these are exactly the cases captured in Equation 5.2. □

Obviously, a tuple (α, β, γ) not satisfying Proposition 3 is an impossible XOR-differential of H. Besides checking whether an XOR-differential is impossible or not, we are also interested in the probability of XOR-differentials. To compute the probability of an XOR-differential with respect to the non-linear operation H of NORX, we can use the following proposition.

Proposition 4. *Let* $\delta = (\alpha, \beta, \gamma)$ *be an* XOR*-differential of the non-linear operation* H *of* NORX*. Its differential probability is then given by*

$$\mathsf{xdp}^{\mathsf{H}}(\delta) = 2^{-\mathsf{hw}((\alpha \vee \beta) \ll 1)} \ .$$

Proof. Without loss of generality we assume that $\alpha \neq 0$ or $\beta \neq 0$. Looking at Equation 5.1 we see that the term $(\alpha \oplus \beta \oplus \gamma)$ has no effect on the probability of the differential δ, since it does not depend on either x or y. It has therefore probability 1.

Analysing the bit level representation of Equation 5.1, we observe that the term

$$(x_{i-1} \wedge \alpha_{i-1}) \oplus (y_{i-1} \wedge \beta_{i-1}) \oplus (\alpha_{i-1} \wedge \beta_{i-1})$$

is balanced, i.e., is 1 with probability 1/2, if $\alpha_{i-1} = 1$ or $\beta_{i-1} = 1$. Under the assumption of independence of α_i and β_i, the overall probability of δ can thus be computed by counting the number of 1s in the first $n - 1$ bits of $\alpha \vee \beta$ or, equivalently, of $(\alpha \vee \beta) \ll 1$ which proves the proposition. □

The above theory can be also expanded to f-differentials as introduced in Section 1.2.2, where $f = \mathsf{H}$. From that we see that a H-differential $(\alpha, \beta) \longrightarrow \gamma$ satisfies the following equation

$$\alpha \oplus \beta \oplus \gamma = ((x \wedge (\alpha \oplus \gamma)) \oplus (y \wedge (\beta \oplus \gamma))) \ll 1 \tag{5.3}$$

for n-bit strings x and y. This formula can be expressed equivalently on bit level as

$$0 = \alpha_0 \oplus \beta_0 \oplus \gamma_0$$
$$0 = (\alpha_i \oplus \beta_i \oplus \gamma_i) \oplus (x_{i-1} \wedge (\alpha_{i-1} \oplus \gamma_{i-1})) \oplus (y_{i-1} \wedge (\beta_{i-1} \oplus \gamma_{i-1})), \quad i > 0 \ .$$

Proposition 5 provides a simple check if a given tuple (α, β, γ) is an (impossible) H-differential, exactly like Proposition 3 for XOR-differentials.

Proposition 5. *Let* H *denote the non-linear operation of NORX. A* H*-differential of the form* $(\alpha, \beta) \longrightarrow \gamma$ *satisfies the following equation:*

$$(\alpha \oplus \beta \oplus \gamma) \wedge (\neg(\gamma \ll 1) \oplus (\alpha \ll 1)) \wedge (\neg(\beta \ll 1) \oplus (\gamma \ll 1)) = 0 \ . \tag{5.4}$$

Proof. It is easy to see that the least significant bits (i.e. $i = 0$) of Equations 5.3 and 5.4 are the same. Therefore, we will consider them no longer. Looking at the bit level representation of Equation 5.3 for $i > 0$, we consider two cases:

- If $\alpha_i \oplus \beta_i \oplus \gamma_i = 0$, then Equation 5.3 has always the solution $x_{i-1} = y_{i-1} = 0$.
- If $\alpha_i \oplus \beta_i \oplus \gamma_i = 1$, then Equation 5.3 is only solvable if either $\alpha_{i-1} \neq \gamma_{i-1}$ or $\beta_{i-1} \neq \gamma_{i-1}$. Furthermore, the bit level representation of Equation 5.4 is given by

  $$(\alpha_i \oplus \beta_i \oplus \gamma_i) \wedge (\alpha_{i-1} \oplus \gamma_{i-1} \oplus 1) \wedge (\beta_{i-1} \oplus \gamma_{i-1} \oplus 1) = 0, \qquad i > 0 \ .$$

  It is evident that the latter equation only holds if $(\alpha_i \oplus \beta_i \oplus \gamma_i) = 0$, $\alpha_{i-1} \neq \gamma_{i-1}$, or $\beta_{i-1} \neq \gamma_{i-1}$. As seen above, these are the very same conditions that define a H-differential.

□

Proposition 6. *Let* H *denote the non-linear operation of NORX and let* $\delta = (\alpha, \beta, \gamma)$ *be a* H*-differential. Its probability is then given by*

$$\mathsf{Hdp}^{\oplus}(\delta) = 2^{-\mathsf{hw}(((\alpha \oplus \gamma) \vee (\beta \oplus \gamma)) \ll 1)} \ .$$

Proof. The claim can be proven analogously to Proposition 4. It follows from the fact that the expression

$$(x_{i-1} \wedge (\alpha_{i-1} \oplus \gamma_{i-1})) \oplus (y_{i-1} \wedge (\beta_{i-1} \oplus \gamma_{i-1}))$$

in the bit level representation of Equation 5.3 is balanced if either $\alpha_{i-1} \oplus \gamma_{i-1} = 1$ or $\beta_{i-1} \oplus \gamma_{i-1} = 1$. □

While we exclusively consider XOR-differentials and -characteristics in the rest of this work, f-differentials might be of interest for future investigations.

Description of NODE

Now that we have introduced the mathematical model, we describe in this part the framework for the search of differential characteristics of a predefined weight. Our tool is freely available at [199] under a public domain-like license. Below, we show the general approach, and refer to Table 29 for the most important CVC code snippets required to describe the search problem.

For modelling the differential propagation through a sequence of operations, we use a technique well known from algebraic cryptanalysis: for every output of an operation a new set of variables is introduced. These output variables are then modelled as a function of their input variables. Moreover, the former are used as input to the next operation. This is repeated until all required operations have been integrated into the problem description. Before we show how the differential propagation in F^{r} is modelled concretely, we introduce the required variables.

Let s denote the number of (column and diagonal) steps to be analysed and let $0 \leq i \leq 15$ and $0 \leq j \leq 2(s-1)$. For example, if we analyse F^2, we have $s = 4$. Let x_i, $y_{i,j}$ and z_i be w-bit sized variables, which model the input, internal and output `XOR` differences of a differential characteristic. Recall that $\mathsf{w} \in \{32, 64\}$ denotes the word size of NORX. Moreover, let $w_{i,k}$, with $0 \leq k \leq s-1$, be w-bit sized helper variables which are used for differential weight computations or equivalently to determine the probability of a differential characteristic. We assume that the probability of a differential characteristic is the sum of weights of each non-linear operation H. Furthermore, let d denote a w-bit sized variable which fixes the total weight of the characteristics we plan to search for. The description of the search problem is generated through the following steps:

1. Every time the function G applies the non-linear operation H we add two expressions to our description:

   a) The equation
   $$\alpha \oplus \beta \oplus \gamma = (\alpha \oplus \beta \oplus \gamma) \wedge ((\alpha \vee \beta) \ll 1)$$
   from Proposition 3, ensures that only non-impossible characteristics are considered. The variables α, β and γ are each substituted by one of the variables x_i, $y_{i,j}$ or z_i.

   b) The equation
   $$w_{i,k} = (\alpha \vee \beta) \ll 1$$
   from Proposition 4 keeps track of the probability of the characteristic. The variables α and β are substituted by the same variables x_i, $y_{i,j}$ or z_i as in the step above.

2. Every time the G function applies a rotation operation we apply the same rotation to the XOR differences, i.e. we add

$$\gamma = (\alpha \oplus \beta) \ggg r$$

to the problem description, with α, β and γ substituted appropriately. Note that the rotation is a linear operation and thus does not change the differential probability. Hence, we do not need to add anything else. Table 29 shows the corresponding CVC code.

3. Add an expression to the description corresponding to the following equation:

$$d = \sum_{k=0}^{s-1} \sum_{i=0}^{15} \mathsf{hw}(w_{i,k}) \ . \tag{5.5}$$

This equation ensures that indeed a characteristic of weight d is found. Depending on the technique how Hamming weights are computed, additional variables might be necessary. Refer to Table 29 for one possible implementation to compute Hamming weights in the CVC language.

4. Set the variable d to the target differential weight and append it to the problem description.

5. Exclude the trivial characteristic mapping an all-zero input difference to an all-zero output difference. To do so, it is sufficient to exclude the all-zero input difference, i.e. to append to the description an expression in CVC language which is equivalent to $\neg\big((x_0 = 0) \wedge ... \wedge (x_{15} = 0)\big)$.

After the generation of the problem description is finished, it can be used to search for differential characteristics using the SMT-solver STP [116]. Alternatively, STP allows to convert the representation of the problem to SMT-LIB2 or CNF, which enables searches with other SMT or SAT solvers, like Boolector [82], Treengeling [52] or CryptoMiniSat [181].

Applications of NODE

In this part, we describe the application of the search framework to the permutation F^r of NORX. Depending on the concrete attack model, there are different ways an attacker could inject differences into the NORX state. During initialisation an adversary is allowed to modify either the nonce words s_1 and s_2 (init_N) or nonce and key words $s_1, s_2, s_4, \ldots, s_7$ ($\mathsf{init}_{N,K}$). During data processing an attacker can inject differences into the words of the rate $s_0, \ldots, s_9$ (rate). Last but not least, we also investigate the case where an attacker

Table 29: CVC code modelling 64-bit operations for differential propagation.

	Differential Validity Check
OP	$\alpha \oplus \beta \oplus \gamma = (\alpha \oplus \beta \oplus \gamma) \wedge ((\alpha \vee \beta) \ll 1)$
CVC	`ASSERT((BVXOR(BVXOR(`α`,`β`),`γ`)) = (BVXOR(BVXOR(`α`,`β`),`γ`) & (((`α` | `β`) ≪ 1)[63:0])));`
	Differential Probability
OP	$w = (\alpha \vee \beta) \ll 1$
CVC	`ASSERT(`w` = (((`α` | `β`) ≪ 1)[63:0]));`
	Cyclic Rotation
OP	$\gamma = (\alpha \oplus \beta) \ggg 8$
CVC	`ASSERT(`γ` = (BVXOR(`α`,`β`) ≫ 8) | ((BVXOR(`α`,`β`) ≪ 56)[63:0]));`
	Hamming Weight
OP	$\mathsf{hw}(w)$, with helper variables $h_0, \ldots, h_5$ and $h_5 = \mathsf{hw}(w)$

CVC	`ASSERT(`m_1` = 0x5555555555555555);`
	`ASSERT(`m_2` = 0x3333333333333333);`
	`ASSERT(`m_4` = 0x0f0f0f0f0f0f0f0f);`
	`ASSERT(`m_8` = 0x00ff00ff00ff00ff);`
	`ASSERT(`m_{16}` = 0x0000ffff0000ffff);`
	`ASSERT(`m_{32}` = 0x00000000ffffffff);`
	`ASSERT(`h_0` = BVPLUS(64,(`w` & `m_1`), (((`w` ≫ 1)[63:0]) & `m_1`)));`
	`ASSERT(`h_1` = BVPLUS(64,(`h_0` & `m_2`), (((`h_0` ≫ 2)[63:0]) & `m_2`)));`
	`ASSERT(`h_2` = BVPLUS(64,(`h_1` & `m_4`), (((`h_1` ≫ 4)[63:0]) & `m_4`)));`
	`ASSERT(`h_3` = BVPLUS(64,(`h_2` & `m_8`), (((`h_2` ≫ 8)[63:0]) & `m_8`)));`
	`ASSERT(`h_4` = BVPLUS(64,(`h_3` & `m_{16}`), (((`h_3` ≫ 16)[63:0]) & `m_{16}`)));`
	`ASSERT(`h_5` = BVPLUS(64,(`h_4` & `m_{32}`), (((`h_4` ≫ 32)[63:0]) & `m_{32}`)));`

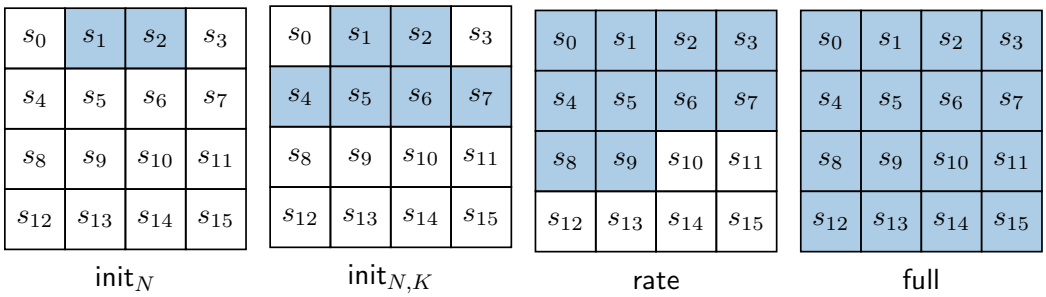


Figure 43: Attacker controllable words (blue) during differential cryptanalysis.

can manipulate the whole state $s_0, \ldots, s_{15}$ (full). Figure 43 gives visual representations of the four settings.

While an attacker is not able to influence the entire state at any point directly due to the duplex construction, the full scenario is nevertheless useful to estimate the general strength of F^{r}, because all of the other settings described above are special cases of the latter. Additionally, it could be useful for chaining characteristics: for example, an attacker could start with a search in the data processing part (i.e. under the rate setting) over a couple of steps, say F^{r_1}, and continue afterwards with a second search starting from the full state for another couple of steps, say F^{r_2}, so that differentials from the second search connect to those from the first. We explore this Divide&Conquer-like strategy in more detail below.

Adapted to r rounds of the permutation F of NORX, we denote a characteristic by

$$\alpha_0 \xrightarrow[w]{\mathsf{F}^{\mathsf{r}}} \alpha_n \;\hat{=}\; \alpha_0 \xrightarrow[w_0]{\mathsf{F}^{r_0}} \ldots\; \alpha_i \xrightarrow[w_i]{\mathsf{F}^{r_i}} \alpha_{i+1} \;\ldots \xrightarrow[w_{n-1}]{\mathsf{F}^{r_{n-1}}} \alpha_n$$

where α_0 is the input, $\alpha_1, \ldots, \alpha_{n-1}$ are internal and α_n is the output difference. The value r_i denotes the number of steps the ith characteristic spans, with $\mathsf{r} = \sum_{i=0}^{n-1} r_i$, and w_i denotes its weight. We assume that the probability of the entire characteristic is equal to the multiplication of probabilities of the partial characteristics. For the total weight of the characteristic it thus holds that $w = \sum_{i=0}^{n-1} w_i$. The notation $\mathsf{F}^{r+0.5}$ describes that we do r full rounds followed by one more column step, e.g. $\mathsf{F}^{1.5}$ corresponds to one full round plus one additional column step.

Experimental Verification of NODE. The goal of the experimental verification is to show that the framework indeed finds valid differentials of a predetermined weight w in F^{r}. Therefore, we used the framework and searched, in the setting full, for differentials

$$\alpha \xrightarrow[w]{\mathsf{F}^{1.5}} \beta$$

with input difference α and output difference β and verified them against a reference implementation of $\mathsf{F}^{1.5}$ written in C. Under these prerequisites our framework found the first differentials at a weight of 12, for both $\mathsf{w} = 32$ and $\mathsf{w} = 64$ which thus should have a probability of about 2^{-12}. To get a better coverage of our verification test, we did not use only differentials of that particular weight, but generated random differentials of weights $w \in \{12, \ldots, 18\}$. These are listed in Table 31. Then we applied them to the C implementation of $\mathsf{F}^{1.5}$ for 2^{w+16} pairs of randomly chosen input states having the input difference of the characteristic. In each case, we checked if the output difference had the predicted pattern. The number of pairs adhering the characteristic should be around 2^{16}. The results are illustrated in Table 30.

In summary, our experiments show that the search framework indeed finds differential paths having the expected properties.

Table 30: Results of the experimental verification of NODE. Notation: w_e expected weight, $\#S$ number of samples, v_e (v_m) expected (measured) value of input/output pairs adhering the differential, w_m measured weight.

			NORX32			NORX64		
w_e	$\#S$	v_e	v_m	$v_m - v_e$	w_m	v_m	$v_m - v_e$	w_m
12	2^{28}	65536	65652	+116	11.997	65627	+91	11.997
13	2^{29}	65536	65788	+252	12.994	65584	+48	12.998
14	2^{30}	65536	65170	−366	14.008	65476	−60	14.001
15	2^{31}	65536	65441	−95	15.002	65515	−21	15.000
16	2^{32}	65536	65683	+147	15.996	65563	+27	15.999
17	2^{33}	65536	65296	−240	17.005	65608	+72	16.998
18	2^{34}	65536	65389	−147	18.003	65565	+29	17.999

Differentials of Weight 0 in G In Section 5.3.1, we derived and verified differentials in G having probability 1 or equivalently weight 0 which was a quite tedious exercise. Now with the help of NODE, we can automatise the task. Additionally, we can even perform a fully automated search to find all differentials of weight 0 in G. It turns out that there are exactly 3 for both the 32- and the 64-bit variant of G. These are listed in Table 32.

Lower Bounds for Weights of Differentials in F^r. We performed an extensive analysis on the weight bounds of differential paths in F^r, where we investigated $1 \leq s \leq 4$ steps for the four different scenarios init_N, $\mathsf{init}_{N,K}$, rate and full. We tried to find the lowest weights where differentials appear for the first time. These cases are listed in Table 33 as entries without brackets. For example, in case of NORX32 under the setting full, there are no differentials in $\mathsf{F}^{1.5}$ with a weight smaller than 12. Entries in brackets are the maximal weights we were capable of examining without finding any differentials.

Due to memory constraints, our methods failed for differential weights higher than those presented in Table 33.

The security of NORX depends heavily on the security of the initialisation which basically transforms the initial state by F^{2r}. As init_N is the most realistic attack scenario, we conducted a search over all possible 1- and 2-bit differences in the nonce words. This search revealed that the best characteristics have weights of 67 (32-bit) and 77 (64-bit) after only one round $\mathsf{F}^{1.0}$, see Table 34. The proofs that there are no differentials of a lower weight were, in some cases, computationally quite expensive. For example, to verify that there are no differentials of weight 66 for NORX32, init_N and F, the SAT-solver Treengeling [52] required about 3 1/2 days of continuous computation on 40 cores and consumed about 15 GiB of memory. Also note that for NORX64, init_N, F, the difference between the verified bound of weight 62 and the best differential characteristic of weight 77 is not too large. Actually, we expect that the bound coincides with the value 77,

Table 31: Characteristics in $\mathsf{F}^{1.5}$ used for experimental verification of NODE.

α				β				w
00000000	00000400	80000080	80000000	00000000	00000000	00000000	80001000	12
00000000	80000400	80000080	00000000	00000000	00000000	00000000	21012100	
00000000	80000000	80808080	80000000	00000000	00000000	00000000	10808080	
00000000	80000000	80800000	80000080	00000000	00000000	00000000	10008080	
80000000	00000000	00000400	80000180	80001000	00000000	00000000	00000000	13
00000000	00000000	80000400	80000080	21012100	00000000	00000000	00000000	
80000000	00000000	80000000	80808080	10808080	00000000	00000000	00000000	
80000080	00000000	80000000	80800000	10008080	00000000	00000000	00000000	
80000080	80000000	00000000	00000400	00000000	80001000	00000000	00000000	14
80000180	00000000	00000000	80000400	00000000	21012100	00000000	00000000	
80808080	80000000	00000000	80000000	00000000	10808080	00000000	00000000	
80800000	80000080	00000000	80000000	00000000	10008080	00000000	00000000	
00000400	80000000	00000400	40100000	00100000	00000000	00000000	00000000	15
80000400	80000000	00000000	00100200	00200021	00000000	00000000	00000000	
80000000	80018000	00000400	00000000	80000010	00000000	00000000	00000000	
80000000	00800000	00040400	40000600	00000010	00000000	00000000	00000000	
00000400	80000080	80000000	00000000	00000000	00000000	80003000	00000000	16
80000400	80000080	00000000	00000000	00000000	00000000	63016100	00000000	
80000000	81808080	80000000	00000000	00000000	00000000	31808080	00000000	
80000000	80800000	80000080	00000000	00000000	00000000	30008080	00000000	
00000000	00000400	80000080	80000000	00000000	00000000	00000000	80001000	17
00000000	80000400	80000080	00000000	00000000	00000000	00000000	21012100	
00000000	80000000	80838780	80000000	00000000	00000000	00000000	10808080	
00000000	80000000	80800000	80000080	00000000	00000000	00000000	10008080	
00000400	00000000	80000000	C0000200	00100000	00000000	00000000	00606001	18
80000400	00000000	00000000	00000200	00200021	00000000	00000000	C24242C0	
80000000	00000000	80000000	00000000	80000010	00000000	00000000	61010160	
80000000	00000000	80000080	C0000000	00000010	00000000	00000000	60010160	

α				β				w
8000000000000000	0000000000000000	0000000000040000	8000000000000080	8000001000000000	0000000000000000	0000000000000000	0000000000000000	12
0000000000000000	0000000000000000	8000000000040000	8000000000000080	2100002001010000	0000000000000000	0000000000000000	0000000000000000	
8000000000000000	0000000000000000	8000000000000000	8000808000000080	1080000000808000	0000000000000000	0000000000000000	0000000000000000	
8000000000000080	0000000000000000	8000000000000000	0080808000000000	1000000000808000	0000000000000000	0000000000000000	0000000000000000	
4000001000000000	0000000000040000	8000000000000000	0000000000040000	0000000000000000	0000100000000000	0000000000000000	0000000000000000	13
0000001000020000	8000000000040000	8000000000000000	0000000000000000	0000000000000000	0000200000000021	0000000000000000	0000000000000000	
0000000000000000	8000000000000000	8000008000000000	0000000000040000	0000000000000000	8000000000000010	0000000000000000	0000000000000000	
4000000000020000	8000000000000000	0000800000000000	0000000004040000	0000000000000000	0000000000000010	0000000000000000	0000000000000000	
0000000000040000	8000000000000080	8000000000000000	0000000000000000	0000000000000000	0000000000000000	8000001000000000	0000000000000000	14
8000000000040000	8000000000000080	0000000000000000	0000000000000000	0000000000000000	0000000000000000	2100002001010000	0000000000000000	
8000000000000000	8003808000000080	8000000000000000	0000000000000000	0000000000000000	0000000000000000	1080000000808000	0000000000000000	
8000000000000000	0080800000000000	8000000000000080	0000000000000000	0000000000000000	0000000000000000	1000000000808000	0000000000000000	
0000000000000000	00000000000C0000	8000000000000080	8000000000000000	0000000000000000	0000000000000000	0000000000000000	8000001000000000	15
0000000000000000	8000000000040000	8000000000000080	0000000000000000	0000000000000000	0000000000000000	0000000000000000	2300006001010000	
0000000000000000	8000000000000000	8000808000000080	8000000000000000	0000000000000000	0000000000000000	0000000000000000	1180000000808000	
0000000000000000	8000000000000000	0080800000000000	8000000000000080	0000000000000000	0000000000000000	0000000000000000	1000000000808000	
0000000000040000	4000001000080000	0000000000040000	8000000000000000	0000000000000000	0000000000000000	0000100000000000	0000000000000000	16
0000000000000000	0000001000020000	8000000000040000	8000000000000000	0000000000000000	0000000000000000	0000200000000021	0000000000000000	
0000000000040000	0000000000000000	8000000000000000	8000008000000000	0000000000000000	0000000000000000	8000000000000010	0000000000000000	
0000000004040000	C00000000E0000	8000000000000000	0000800000000000	0000000000000000	0000000000000000	0000000000000010	0000000000000000	
8000000000000080	8000000000000000	0000000000000000	0000000000040000	0000000000000000	8000007000000000	0000000000000000	0000000000000000	17
8000000000000080	0000000000000000	0000000000000000	8000000000040000	0000000000000000	E300006001010000	0000000000000000	0000000000000000	
8000808000000180	8000000000000000	0000000000000000	8000000000000000	0000000000000000	7180000000808000	0000000000000000	0000000000000000	
0080808000000000	8000000000000080	0000000000000000	8000000000000000	0000000000000000	7000000000808000	0000000000000000	0000000000000000	
0000000000040000	8000000000000000	0000000000040000	400000F000000000	0000100000000000	0000000000000000	0000000000000000	0000000000000000	18
8000000000040000	8000000000000000	0000000000000000	0000001000020000	0000200000000021	0000000000000000	0000000000000000	0000000000000000	
8000000000000000	8000008000000000	0000000000040000	0000000000000000	8000000000000010	0000000000000000	0000000000000000	0000000000000000	
8000000000000000	0000800000000000	000000000C040000	4000000000020000	0000000000000010	0000000000000000	0000000000000000	0000000000000000	

Table 32: Characteristics of weight 0 in G.

Characteristics (32-bit)					Characteristics (64-bit)				
α	80000000	80000000	80000000	00000000	α	8000000000000000	8000000000000000	8000000000000000	0000000000000000
β	00000000	00000001	80000000	00000000	β	0000000000000000	0000000000000001	8000000000000000	0000000000000000
α	80000000	00000000	80000000	80000080	α	8000000000000000	0000000000000000	8000000000000000	8000000000000080
β	80000000	00000000	00000000	00000000	β	8000000000000000	0000000000000000	0000000000000000	0000000000000000
α	00000000	80000000	00000000	80000080	α	0000000000000000	8000000000000000	0000000000000000	8000000000000080
β	80000000	00000001	80000000	00000000	β	8000000000000000	0000000000000001	8000000000000000	0000000000000000

Table 33: Lower bounds for differential trail weights.

	NORX32				NORX64			
	init_N	$\mathsf{init}_{N,K}$	rate	full	init_N	$\mathsf{init}_{N,K}$	rate	full
$\mathsf{F}^{0.5}$	6	2	2	0	6	2	2	0
$\mathsf{F}^{1.0}$	67	22	10	2	(62)	22	12	2
$\mathsf{F}^{1.5}$	(60)	(40)	(31)	12	(53)	(35)	(27)	12
$\mathsf{F}^{2.0}$	(61)	(45)	(34)	(28)	(51)	(37)	(30)	(23)

similarly as in the case of NORX32.

Extrapolating the above results to F^8 (i.e. $\mathsf{r} = 4$), we get lower weights of $61+3\cdot 28 = 145$ (init_N) or $45 + 3 \cdot 28 = 129$ ($\mathsf{init}_{N,K}$) for NORX32 and $51 + 3 \cdot 23 = 132$ (init_N) or $37 + 3 \cdot 23 = 106$ ($\mathsf{init}_{N,K}$) for NORX64. However, these are obviously rather loose bounds and we expect the real ones to be considerably higher.

Table 34: Characteristics in NORX initialisation (init_N) after F.

Characteristic of weight 67 (32-bit)					Characteristic of weight 76 (64-bit)				
α	00000000	80008000	00000000	00000000	α	0000000000000000	8000800000000000	0000000000000000	0000000000000000
	00000000	00000000	00000000	00000000		0000000000000000	0000000000000000	0000000000000000	0000000000000000
	00000000	00000000	00000000	00000000		0000000000000000	0000000000000000	0000000000000000	0000000000000000
	00000000	00000000	00000000	00000000		0000000000000000	0000000000000000	0000000000000000	0000000000000000
β	00000000	12001200	02300210	00020002	β	0000000000000000	1200120000000000	0230021000000000	0002000200000000
	20002000	00000000	20402840	084203c2		2000200000000000	0000000000000000	2040284000000000	084203c200000000
	063103f1	10021002	00000000	12201620		063103f100000000	1002100200000000	0000000000000000	1220162000000000
	12101210	820082c1	00020002	00000000		1210121000000000	820082c100000000	0002000200000000	0000000000000000

Differential Characteristics in F^4. This part shows how we constructed differential characteristics in F^4 under setting full for both versions of the permutation, i.e. 32- and

64-bit. To be more precise, the trails are of the form

$$\alpha_0 \xrightarrow[w_0]{\mathsf{F}} \alpha_1 \xrightarrow[w_1]{\mathsf{F}} \alpha_2 \xrightarrow[w_2]{\mathsf{F}} \alpha_3 \xrightarrow[w_3]{\mathsf{F}} \alpha_4 \ .$$

Unsurprisingly, a direct approach to find such characteristics turned out to be infeasible. Hence, we decomposed the search into multiple parts and constructed the entire path step by step.

Initially, we conducted searches that only stretched over $\mathsf{r} \leq 2$ rounds. After tens of thousands of iterations using many different search parameter combinations we found differentials having internal differences of Hamming weight 1 and 2 after one application of F. We identify these characteristics by δ_0 and δ_1 below. We also used a probability-1 differential in G, which are listed as the first entry in the Table 32, as a starting point. The latter is denoted by δ_2. We expanded each partial characteristic, for both word sizes, in forward and backward direction one column or diagonal step at a time, until their paths stretched the entire 4 rounds. The concrete representations of the characteristics and their respective weights are depicted in Tables 35 (32-bit) and 36 (64-bit), respectively. The best characteristics we found this way for 32- and 64-bit F^4 have weights of 584 and 836, respectively. These weights translate to probabilities of 2^{-584} and 2^{-836}, which are orders of magnitude beyond any feasible attack, besides the fact that they are not applicable to the NORX scheme anyway. However, those results give a first impression on the resistance of F^r against differential attacks. Recall that the NORX initialisation uses at least twice the amount of rounds, i.e. F^8 compared to the presented characteristics in this part. The search for these differentials brought our methods to the limits of computational practicability, especially for the 64-bit case. It was not possible to stretch the paths any further.

Iterative Differential Characteristics in F. We also performed extensive searches for iterative differential characteristics in F, i.e. trails of the form

$$\alpha \xrightarrow[w]{\mathsf{F}} \beta$$

with $\alpha = \beta$. Using NODE, we could show that there are no such differentials up to a weight w of 29 (32-bit) and 27 (64-bit), before our methods failed due to computational constraints. Extrapolating these results to F^8 and F^{12}, i.e. the number of initialisation rounds for $\mathsf{r} = 4$ and $\mathsf{r} = 6$, we get lower weight bounds of 232 and 348, for 32-bit, or of 216 and 324 for 64-bit. The best iterative differentials we could find for F have weights of 512 (32-bit) and 843 (64-bit) and are depicted in Table 37. These weights are obviously much higher than our guaranteed lower bounds, and we therefore expect that the latter are also much better compared to the values we were able to verify computationally.

Table 35: Differential characteristics in F^4 (32-bit).

Characteristic δ_0									
α_0				w_0	α_1				w_1
80140100	90024294	84246020	92800154	172	40100000	00000400	80000000	00000400	11
e4548300	52240214	e0202424	d0004054		00100200	80000400	80000000	00000000	
c4464046	00a08480	c1008108	90d43134		00000000	80000000	80008000	00000400	
e200c684	e2eac480	a4848881	06915342		40000200	80000000	00800000	00040400	
α_2				w_2	α_3				w_3
00000000	00000000	00000000	00000000	44	04042425	00100002	00020000	02100000	357
00000000	00000000	00000000	00000000		04200401	42024200	20042024	20042004	
00000000	80000000	00000000	00000000		10001002	80000200	25250504	10021010	
00000000	00000000	00000000	00000000		10020010	00001002	00000210	04252504	
α_4									
c4001963	804da817	0c05b60e	12220503	total weight: **584**					
9072b909	185b792a	cc0d56cd	7e0ac646						
80116300	100c2800	8f003320	3b270222						
01056104	88000041	92002824	04210001						

Characteristic δ_1									
α_0				w_0	α_1				w_1
5828126e	12523a05	84960644	c66a0440	206	84000480	80800000	80000000	40000600	19
dca2126c	12501886	04168404	666a4440		04008480	00800000	00000000	80000600	
41528094	00848446	00800400	430ac506		00800080	80800080	80000000	80000000	
d648d4cb	c4868483	000482c0	f8818643		00808000	80008080	80000080	c0000000	
α_2				w_2	α_3				w_3
00000000	80000000	00000000	00000000	26	20010020	80001000	10000000	00000010	344
00000000	00000000	00000000	00000000		00210020	42404242	21012100	21002100	
00000000	00000000	00000000	00000000		00801080	00108000	01202101	10808080	
00000000	80000000	00000000	00000000		10008080	00801000	00100000	01202001	
α_4									
8150c742	2c36b006	33202401	260ea0a5	total weight: 595					
02cf014b	d93018b9	860ecacd	64464804						
01802004	e6164858	23020245	06866087						
50802042	94122c04	20110121	00a10002						

Characteristic δ_2									
α_0				w_0	α_1				w_1
482304cd	c4bc4096	1000012b	7012c114	266	80080500	80000000	40000000	80100280	27
c8a7008d	c69c8216	9752612b	f2538210		80080400	80000000	c0000400	80100280	
a915cc80	8480c200	0014a080	0e84500c		00000000	80008000	c0004400	c0808280	
04448561	0286c2c4	b5f7e220	46164d22		00000180	00800000	80440400	80820040	
α_2				w_2	α_3				w_3
80000000	00000000	00000000	00000000	12	00000000	00000000	00100000	00202001	286
80000000	00000000	00000000	00000000		42424240	00000000	00000000	00200021	
80000000	00000000	00000000	00000000		80000010	21010120	00000000	00000000	
00000000	00000000	00000000	00000000		00000000	00000010	20010120	00000000	
α_4									
06060a24	26426628	14160221	882c28a0	total weight: 591					
51ddd93d	631c46c9	7cda4426	09131b01						
6eeae416	91000b60	380d0212	04898008						
0a262202	8a060060	1303200d	0ca48400						

Table 36: Differential characteristics in F^4 (64-bit).

Characteristic δ_0									
α_0				w_0	α_1				w_1
4400445880011086	0080303002202404	6280500400041142	0440803004010000	202	8000000000000000	0000000000040000	4000001000000000	0000000000040000	11
c600841880821086	0480302000242400	c0a0505000041142	c44080b804020000		8000000000000000	0000000000000000	0000001000020000	8000000000040000	
840080c080828000	0404042004040404	c140128800500842	0200805004860004		8000008000000000	0000000000040000	0000000000000000	8000000000000000	
820000c006038044	0004201402040404	e262c05453080001	c000d088840704c2		0000800000000000	0000000004040000	4000000000020000	8000000000000000	
α_2				w_2	α_3				w_3
0000000000000000	0000000000000000	0000000000000000	0000000000000000	45	0000020000000000	0002100000000000	0004042004040021	0000100000020000	615
0000000000000000	0000000000000000	0000000000000000	0000000000000000		2000042000040020	2000040000200004	0004200000000401	420040004a4a0808	
0000000000000000	0000000000000000	0000000000000000	8000000000000000		2104042021210404	1000021000000010	1000000000100002	8000000000000200	
0000000000000000	0000000000000000	0000000000000000	0000000000000000		0000000000000210	0004040021210404	1000020000000010	0000000000100002	
α_4									
123ca460842bb1bd	145c6e810d01c220	21e5a1c48ea44418	430242b2434603b2	total weight: 873					
87b140498a09f28c	90d451e2c61e4651	c065d8f8994bd83f	900929019a033431						
08c14804c0011002	8c0a6270676c6304	c0b08c5644098836	c146de088200c830						
20c0a4220101800	900e012022040002	8060013428002420	8100100001408000						

Characteristic δ_1									
α_0				w_0	α_1				w_1
44846c6c4294a646	12064410c24d80b7	0480801006060044	a26040440a040040	293	8400000000040080	8006800000000000	8000000000000000	4000000000a0000	30
80c46c6c42102e44	920646908 20c80b6	8480009006040004	026040080a040040		0400008000040380	0001800000000000	0000000000000000	8000000000060000	
008344120a00d005	0282048202c48002	8080808080840080	470000b80a40004a		0000800000000280	8000800000800000	8000000000000000	8000000000000000	
c704d2128654c900	0244840284c282c1	80808000840280c0	a040984c41080a47		8080818000000000	8000000080000080	8000000000000080	4000000000000000	
α_2				w_2	α_3				w_3
0000000000000000	8000000000000000	0000000000000000	0000000000000000	26	2000000001000020	8000001000000000	1000000000000000	0000000000000010	531
0000000000000000	0000000000000000	0000000000000000	0000000000000000		0000000021000020	4002020040420040	2100002001010000	2000010000210000	
0000000000000000	0000000000000000	0000000000000000	0000000000000000		0000008000108000	0000000010800000	0001010020210000	1080000000808000	
0000000000000000	8000000000000000	0000000000000000	0000000000000000		1000000000808000	0000008000100000	0000000010000000	0001010020200000	
α_4									
a2ac9480a1922800	5a1247c080d3304b	a21f36c081672c51	040c408687216204	total weight: 880					
257c5952c9292742	3dd0a895b72a540c	fe5689044edc891f	5bd80b07079ac635						
5006982c2016bf03	dee01500dbd4020f	3220700261644022	8988400307cc4326						
b020941801008280	c0801308091a0249	d240021440004815	8484200206242020						

Characteristic δ_2									
α_0				w_0	α_1				w_1
00900824010288c5	4000443880011086	2240120442 20ac43	e004044484049520	349	8000000800050000	8000000000000000	4000000000000000	0000001000020080	27
4080882001010885	4600841880821086	a3c0721444632c43	c224440007849504		8000000800040000	8000000000000000	c000000000040000	8000001000020080	
81600850830b0484	840080c080868000	8004449040c14400	8102101840908a80		0000000000000000	8000008000000000	c000004000040000	4000808000020080	
6191548c08000581	0200004006038044	8104f01c8702c0e0	60605084938886e3		0000000000010080	0000800000000000	8000400004040000	80808000020000c0	
α_2				w_2	α_3				w_3
8000000000000000	0000000000000000	0000000000000000	0000000000000000	12	0000000000000000	0000000000000000	0000100000000000	0000202000000001	448
8000000000000000	0000000000000000	0000000000000000	0000000000000000		4200404002020040	0000000000000000	0000000000000000	0000200000000021	
8000000000000000	0000000000000000	0000000000000000	0000000000000000		8000000000000010	2100000001010020	0000000000000000	0000000000000000	
0000000000000000	0000000000000000	0000000000000000	0000000000000000		0000000000000000	0000000000000010	2000000001010020	0000000000000000	
α_4									
321a4500060e4e2e	27404405026e500e	3806422387200a08	8c40f4a0884c0820	total weight: **836**					
71540fb858cb9902	ee018cc282747980	c714164174ce3eb9	1a49a091101191e1						
786680d0e46406cb	14440844013274e6	03a843203f071b7c	09a840c00c0ccc78						
4000404a22120005	07220c4202016240	2aa4200a0a041a62	84a468682000601c						

Table 37: Iterative differential characteristics F.

	Characteristic, 32-bit, weight 512			
$\alpha = \beta$	818c959b	00186049	eb5b7984	791c6da1
	677b513d	80000400	00000227	5293655f
	00809a2b	bfa98bff	c08b8e89	0000711c
	800027c3	f984eb5b	6d81f915	b5aaa99d

	Characteristic, 64-bit, weight 843			
$\alpha = \beta$	0000000100000000	0000000000000000	f77c78b200000d04	0000000000000000
	be7fffeffe0f349f	0000000000000000	6c07fbd200000001	ff1ab5be4e7500be
	0060c54927018000	0000000000000000	0000000000000000	b603fde900000000
	b6035caf00000000	0000000000000000	0000000000000000	0000000000000000

Differential Characteristics with Equal Columns in F. The class of weak states from Section 5.2.2 can be transformed into differential trails having four equal columns. The best characteristics we found for one round have a weight of 44 for both 32-bit and 64-bit. To find them, we used the already well known probability-1 ($w = 0$) differential in G that is shown in Table 27. Recall, that this particular differential was also used in the construction of the differential path v_3 for F^4 as was shown above. Concrete representations of these differentials are given in Table 38.

Table 38: Differential characteristics with equal columns F.

	Characteristic, 32-bit, weight 44			
α	80000000	80000000	80000000	80000000
	80000000	80000000	80000000	80000000
	80000000	80000000	80000000	80000000
	00000000	00000000	00000000	00000000
β	00102001	00102001	00102001	00102001
	42624221	42624221	42624221	42624221
	a1010110	a1010110	a1010110	a1010110
	20010110	20010110	20010110	20010110

	Characteristic, 64-bit, weight 44			
α	8000000000000000	8000000000000000	8000000000000000	8000000000000000
	8000000000000000	8000000000000000	8000000000000000	8000000000000000
	8000000000000000	8000000000000000	8000000000000000	8000000000000000
	0000000000000000	0000000000000000	0000000000000000	0000000000000000
β	0000102000000001	0000102000000001	0000102000000001	0000102000000001
	4200604002020021	4200604002020021	4200604002020021	4200604002020021
	a100000001010010	a100000001010010	a100000001010010	a100000001010010
	2000000001010010	2000000001010010	2000000001010010	2000000001010010

Applications to Other Ciphers

The techniques presented in this section are obviously not restricted to NORX only. In principle, every function based on integer addition, as shown for Salsa20 in [187], and/or bitwise logical operations, like OR, NAND, NOR and so on, can be analysed just as easily. For LRX ciphers, all one has to do is to rewrite their non-linear operations in terms of bitwise logical AND which then allows to reuse the results from above.

We used a modified version of this framework to attack McMambo [175] (LRX) and Wheesht [184] (ARX), two other CAESAR candidates. As reported to the competition's mailing list, we found iterative differentials of low-probability which enable practical forgery attacks on these ciphers.

5.4 Rotational Cryptanalysis

In the following, we present some rotational properties of the NORX permutation F^{r} and we derive bounds against a few simple rotational attacks. For more general information on rotational cryptanalysis, see Section 1.2.5.

Let f be a vector Boolean function $f : \mathbb{F}^{2n} \longrightarrow \mathbb{F}^n$ and let x, y be n-bit strings. We call $(x, y) \in \mathbb{F}^{2n}$ a *rotational pair* with respect to f if the equation

$$f(x, y) \ggg r = f(x \ggg r, y \ggg r)$$

holds for a rotation offset r.

Proposition 7. *Let* H *be the non-linear function of NORX, and let* x, y *be* n*-bit strings, and let* r *be a rotation offset. The probability of* $(x, y) \in \mathbb{F}^{2n}$ *being a rotational pair is:*

$$\Pr(\mathsf{H}(x, y) \ggg r = \mathsf{H}(x \ggg r, y \ggg r)) = \frac{9}{16} \ (\approx 2^{-0.83}) \ .$$

Proof. After evaluating and simplifying the equation

$$\mathsf{H}(x, y) \ggg r = \mathsf{H}(x \ggg r, y \ggg r)$$

we get

$$((x \wedge y) \ll 1) \ggg r = ((x \ggg r) \wedge (y \ggg r)) \ll 1$$

Translating this equation to bit vectors we obtain

$$\begin{aligned} &(x_{r-1} \wedge y_{r-1}, \ldots, x_0 \wedge y_0, 0, x_{n-2} \wedge y_{n-2}, \ldots, x_r \wedge y_r) \\ = &(x_{r-1} \wedge y_{r-1}, \ldots, x_0 \wedge y_0, x_{n-1} \wedge y_{n-1}, x_{n-2} \wedge y_{n-2}, \ldots, 0) \ . \end{aligned}$$

The probability that those two vectors match is $(3/4)^2 = 9/16$, as $a \wedge b = 0$ with probability $3/4$ for bits a and b chosen uniformly at random. □

Now we can use Proposition 7 and Theorem 1 from [153], under the assumption that the latter also holds for H, to compute the probability of $\Pr(\mathsf{F}^{\mathsf{r}}(S) \ggg r = \mathsf{F}^{\mathsf{r}}(S \ggg r))$ for a state S, a number of rounds r and a rotation offset r. It is given by:

$$\Pr(\mathsf{F}^{\mathsf{r}}(S) \ggg r = \mathsf{F}^{\mathsf{r}}(S \ggg r)) = (9/16)^{4 \cdot 4 \cdot 2 \cdot \mathsf{r}} \ .$$

Table 39 summarises the (rounded) weights (i.e. the negative logarithms of the probabilities Pr) for different values of r, which are relevant for NORX.

As a consequence, the permutation F^{r} on a $16W$ state is indistinguishable from a random permutation for $\mathsf{r} \geq 20$ if $\mathsf{w} = 32$ and for $\mathsf{r} \geq 39$ if $\mathsf{w} = 64$ with probabilities of $\Pr \leq 2^{-531}$ and $\Pr \leq 2^{-1035}$, respectively.

Let f be a vector Boolean function $f : \mathbb{F}^{2n} \longrightarrow \mathbb{F}^n$ and let x, y be n-bit strings. We call (x, y) a *rotational fix point* with respect to f and a rotation offset r if the following equation holds:

$$f(x, y) \ggg r = f(x, y) \ .$$

Table 39: Weights for rotational distinguishers of F^r.

r	4	6	8	12
w	106	159	212	318

Proposition 8. *Let f be a vector Boolean function $f : \mathbb{F}^{2n} \longrightarrow \mathbb{F}^n,\ (x, y) \mapsto f(x, y)$, which is a permutation on $\mathbb{F}^n$, if either x or y is fixed. The probability that $(x, y) \in \mathbb{F}^{2n}$ is a rotational fix point with respect to f and a rotation offset r is:*

$$\Pr(f(x, y) \ggg r = f(x, y)) = 2^{-(n-\gcd(r,n))} .$$

Proof. The first important observation is that the statement of this proposition is independent of the function f, as it only makes a claim on the image of f. Thus, it is sufficient to prove the proposition for $z \ggg r = z$, where $z = f(x, y)$ and x or y was fixed.

We identify the indices of an n-bit string by the elements of the group $G := \mathbb{Z}_n$. Let

$$\tau : G \longrightarrow G,\ i \bmod n \mapsto (i + 1) \bmod n .$$

Then τ obviously generates the cyclic group G, i.e. $\mathrm{ord}(\tau) = n$. Moreover, for an arbitrary $r \in \mathbb{Z}$ we have

$$\mathrm{ord}(\tau^r) = \frac{n}{\gcd(r, n)}$$

see [227, Section 6.2]. In other words, the subgroup $H := \langle \tau^r \rangle$ of G has order $n/\gcd(r, n)$.

By Lagrange's theorem we have $\mathrm{ord}(G) = [G : H] \cdot \mathrm{ord}(H)$ and it follows for the group index $[G : H] = \gcd(r, n)$ which corresponds to the number of (left) cosets of H in G. These cosets contain the indices of a bit string which are mapped onto each other by the cyclic rotation $\ggg r$. This means that there are $2^{\gcd(r,n)}$ n-bit strings z which satisfy $z \ggg r = z$. Thus, the probability that an n-bit string z, chosen uniformly at random among all n-bit strings, satisfies $z \ggg r = z$ is $2^{-(n-\gcd(r,n))}$. This proves the proposition. □

A direct consequence of Proposition 8 is that for n even and $r = n/2$ the probability that (x, y) is a rotational fix point is $2^{-n/2}$. The rotation $r = n/2$, which swaps the two halves of a bit string, is especially interesting for cryptanalysis as it results in the highest probability among all $0 < r < n$.

The non-linear function H of NORX obviously satisfies the requirement of being a permutation on $\mathbb{F}^n$, if one of its inputs is fixed. Therefore, we get probabilities of 2^{-16} (32-bit, $r = 16$) and 2^{-32} (64-bit, $r = 32$), that (x, y) is a rotational fix point of H.

NORX includes several defence mechanisms to increase the difficulty of finding exploitable rotation-invariant behaviour and to prevent the transfer of potential rotational attacks from the permutation to the full cipher:

- During initialisation 10 out of 16 words are loaded with asymmetric constants which impedes the occurrence of rotation-invariant behaviour and limits the freedom of an attacker. A similar approach is also used in Salsa20 [36].
- The non-linear operation H contains a non-rotational-invariant bit-shift $\ll 1$.
- The duplex construction of NORX prevents an attacker from modifying the complete internal state at a given time. He is only able to influence the rate bits, i.e. at most 10w bits of the state, and has to "guess" the other 6w bits in order to mount an attack.

5.5 Conclusion

This section presents first results on the cryptanalysis of NORX. We started with some general observations on fix points, weak states, algebraic properties and slide attacks, and discussed for each case why it is unlikely that an adversary can exploit the property to mount an attack on NORX.

In the next part, we studied differential properties of the core permutation F^r of NORX and derived first bounds on the complete scheme. We started with the analysis of the propagation of 1-bit differences through the G permutation and used that knowledge to construct simple differentials on a few steps of F. This approach quickly becomes infeasible, especially when analysing a larger number of steps of F. We therefore developed a theory to describe differential propagation in H which can be extended to F^r and allows to search for differentials and characteristics in the permutation F^r of NORX. All mathematical claims are verified by rigorous proofs. Afterwards, we introduced NODE, the NORX Differential Search Engine, an implementation of the above techniques which allows to automate the search for differential paths using the power of SMT- and SAT-solvers like STP, Boolector, Treengeling, and CryptoMiniSat.

We continued with a discussion of our extensive experiments using NODE, where we analysed various differential properties of G and F. We explained the approach how we constructed differentials in F^4, how we derived lower bounds for F^r, with $0 \leq \mathsf{r} \leq 2$, and how we analysed various other types, like iterative and equal-column differentials. From that, we conclude that there is a large gap between those bounds of differential weights that are computationally verifiable and the weights of the best differentials that we were able to find. In particular, we showed that the probabilities of differential characteristics in initialisation are bounded by 2^{-66} (NORX32) and 2^{-62} (NORX64) after only one round F for an attacker who has control over the nonce words. Injecting differences over the nonce is one of the realistic attack scenarios when targeting NORX, since the nonce is publicly known. The first characteristic we found for NORX32 has a probability of 2^{-67} which connects seamlessly to the above bound. For NORX64, though, the first characteristic we found in the above scenario has a probability of 2^{-77}, meaning there is still a gap to

the confirmed value of 2^{-62}. However, we expect that in the case of NORX64 it works out in the same way as for NORX32. On the other hand, the best differentials in F^4 have probabilities of 2^{-584} (32-bit) and 2^{-836} (64-bit) for an attacker who has control over the full state. These values are obviously far beyond any serious attack on NORX. Additionally, an attacker has at no point during data processing full control over the entire NORX state. Thus, initialisation with F^8 ($\mathsf{r} = 4$) and F^{12} ($\mathsf{r} = 6$) seems to have a high security margin against differential attacks.

For rotational cryptanalysis, we analysed some of the rotational properties of F^r and were able to derive lower weight bounds of 212 and 318 for distinguishers on F^8 and F^{12} using a mix of new and already known results. We stress that these distinguishers currently only hold for the bare permutation F^r and can not be transferred to the complete scheme directly. This is due to the fact, that they do not take into consideration the additional protection provided by the duplex construction of NORX or the asymmetric constants used during state initialisation.

Bibliography

[1] D. Agrawal, B. Archambeault, J. R. Rao, and P. Rohatgi. The EM Side-Channel(s). *Cryptographic Hardware and Embedded Systems — CHES 2002*. Volume **2523**, Lecture Notes in Computer Science. Springer, 2003.

[2] M. Ågren, M. Hell, T. Johansson, and W. Meier. Grain-128a: A New Version of Grain-128 with Optional Authentication. *International Journal of Wireless and Mobile Computing*, **5**(1), 2011.

[3] M. R. Albrecht, N. T. Courtois, D. Hulme, and G. Song. Bit-Slice Implementation of PRESENT in Pure Standard C, v1.5. 2011. Opensource code available at `https://bitbucket.org/malb/research-snippets/src`.

[4] M. R. Albrecht, K. G. Paterson, and G. J. Watson. Plaintext Recovery Attacks Against SSH. *Proceedings of the 30th IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2009.

[5] N. AlFardan, D. J. Bernstein, K. G. Paterson, B. Poettering, and J. C. N. Schuldt. On the Security of RC4 in TLS. *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. USENIX, 2013.

[6] R. Anderson, E. Biham, and L. Knudsen. Serpent: A Proposal for the Advanced Encryption Standard. 1998. `http://www.cl.cam.ac.uk/~rja14/Papers/serpent.pdf`.

[7] E. Andreeva, J. Daemen, B. Mennink, and G. V. Assche. Security of Keyed Sponge Constructions Using a Modular Proof Approach. *Fast Software Encryption — FSE 2015*. Springer, 2015. (to appear).

[8] F. Armknecht. On the Existence of Low-Degree Equations for Algebraic Attacks. IACR Cryptology ePrint Archive, Report 2004/185. 2004. `http://eprint.iacr.org/2004/185`.

[9] ARM ® NEON™ General-purpose SIMD Engine. ARM Holdings, 2014. `http://www.arm.com/products/processors/technologies/neon.php`.

[10] F. Arnault, T. Berger, and C. Lauradoux. F-FCSR Stream Ciphers. *New Stream Cipher Designs*. Volume **4986**, Lecture Notes in Computer Science. Springer, 2008.

[11] J. Arndt. Matters Computational: Ideas, Algorithms, Source Code. Springer, 2010.

[12] J.-P. Aumasson and D. J. Bernstein. SipHash: a Fast Short-input PRF. IACR Cryptology ePrint Archive, Report 2012/351. 2012. http://eprint.iacr.org/2012/351.

[13] J.-P. Aumasson, I. Dinur, L. Henzen, W. Meier, and A. Shamir. Efficient FPGA Implementations of High-Dimensional Cube Testers on the Stream Cipher Grain-128. IACR Cryptology ePrint Archive, Report 2009/218. 2009. http://eprint.iacr.org/2009/218.

[14] J.-P. Aumasson, S. Fischer, S. Khazaei, W. Meier, and C. Rechberger. New Features of Latin Dances: Analysis of Salsa, ChaCha and Rumba. *Fast Software Encryption — FSE 2008*. Volume **5086**, Lecture Notes in Computer Science. Springer, 2008.

[15] J.-P. Aumasson, L. Henzen, W. Meier, and M. Naya-Plasencia. QUARK: A Lightweight Hash. *Journal of Cryptology*, **26**(2), 2013.

[16] J.-P. Aumasson, L. Henzen, W. Meier, and R. C.-W. Phan. SHA-3 Proposal BLAKE. *NIST SHA-3 Proposal*, 2010. https://131002.net/blake.

[17] J.-P. Aumasson and P. Jovanovic. CAESAR and NORX — The Future of Authenticated Encryption? 31st Chaos Communication Congress. 2014. http://media.ccc.de/browse/congress/2014/31c3_-_6137_-_en_-_saal_g_-_201412291600_-_caesar_and_norx_-_philipp_jovanovic_-_aumasson.html.

[18] J.-P. Aumasson, P. Jovanovic, and S. Neves. NORX8 and NORX16: Authenticated Encryption for Low-End Systems. Trustworthy Manufacturing and Utilization of Secure Devices — TRUDEVICE 2015.

[19] J.-P. Aumasson, P. Jovanovic, and S. Neves. Analysis of NORX: Investigating Differential and Rotational Properties. *Progress in Cryptology — LATINCRYPT 2014*. Volume **8895**, Lecture Notes in Computer Science. Springer, 2014.

[20] J.-P. Aumasson, P. Jovanovic, and S. Neves. NORX: Parallel and Scalable AEAD. *European Symposium on Research in Computer Security — ESORICS 2014*. Volume **8713**, Lecture Notes in Computer Science. Springer, 2014.

[21] J.-P. Aumasson, S. Neves, Z. Wilcox-O'Hearn, and C. Winnerlein. BLAKE2: Simpler, Smaller, Fast as MD5. *Applied Cryptography and Network Security — ACNS 2013*. Volume **7954**, Lecture Notes in Computer Science. Springer, 2013.

[22] S. Babbage and M. Dodd. The MICKEY Stream Ciphers. *New Stream Cipher Designs*. Volume **4986**, Lecture Notes in Computer Science. Springer, 2008.

[23] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The Sorcerer's Apprentice Guide to Fault Attacks. Volume **94**, (2), 2006.

[24] G. V. Bard. Algebraic Cryptanalysis. Springer, 2009.

[25] G. V. Bard, N. T. Courtois, and C. Jefferson. Efficient Methods for Conversion and Solution of Sparse Systems of Low-Degree Multivariate Polynomials over GF(2) via SAT-Solvers. Cryptology ePrint Archive, Report 2007/024. 2007. http://eprint.iacr.org/2007/024.

[26] R. J. Bayardo Jr. and R. C. Schrag. Using CSP Look-back Techniques to Solve Real-world SAT Instances. *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence*. AAAI'97/IAAI'97. AAAI Press, 1997.

[27] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers. The Simon and Speck Families of Lightweight Block Ciphers. IACR Cryptology ePrint Archive, Report 2013/404. 2013. http://eprint.iacr.org/2013/404.

[28] M. Beeler, R. W. Gosper, and R. Schroeppel. HAKMEM. Artificial Intelligence Memo (239). Massachusetts Institute of Technology, 1972. http://dspace.mit.edu/handle/1721.1/6086.

[29] C. Beierle, P. Jovanovic, M. M. Lauridsen, G. Leander, and C. Rechberger. Analyzing Permutations for AES-like Ciphers: Understanding ShiftRows. *Topics in Cryptology — CT-RSA 2015*. Volume **9048**, Lecture Notes in Computer Science. Springer, 2015.

[30] M. Bellare, R. Canetti, and H. Krawczyk. Keying Hash Functions for Message Authentication. *Advances in Cryptology — CRYPTO 1996*. Volume **1996**, Lecture Notes in Computer Science. Springer, 1996.

[31] M. Bellare, T. Kohno, and C. Namprempre. Breaking and Provably Repairing the SSH Authenticated Encryption Scheme: A Case Study of the Encode-then-Encrypt-and-MAC Paradigm. *ACM Transactions on Information and System Security*, **7**(2), 2004.

[32] M. Bellare and C. Namprempre. Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm. *Advances in Cryptology — ASIACRYPT 2000*. Volume **1976**, Lecture Notes in Computer Science. Springer, 2000.

[33] M. Bellare, P. Rogaway, and D. Wagner. The EAX Mode of Operation. *Fast Software Encryption — FSE 2004*. Volume **3017**, Lecture Notes in Computer Science. Springer, 2004.

[34] C. Berbain, O. Billet, A. Canteaut, N. Courtois, H. Gilbert, L. Goubin, A. Gouget, L. Granboulan, C. Lauradoux, M. Minier, T. Pornin, and H. Sibert. SOSEMANUK, a Fast Software-Oriented Stream Cipher. *New Stream Cipher Designs*. Volume **4986**, Lecture Notes in Computer Science. Springer, 2008.

[35] D. J. Bernstein. Cache-Timing Attacks on AES. 2005. http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[36] D. J. Bernstein. Salsa20 Security. 2005. http://cr.yp.to/snuffle/security.pdf.

[37] D. J. Bernstein. The Poly1305-AES Message-Authentication Code. *Fast Software Encryption — FSE 2005*. Volume **3557**, Lecture Notes in Computer Science. Springer, 2005.

[38] D. J. Bernstein. Understanding Brute Force. 2005. http://cr.yp.to/snuffle/bruteforce-20050425.pdf.

[39] D. J. Bernstein. QHASM SOFTWARE PACKAGE. 2007. http://cr.yp.to/qhasm.html.

[40] D. J. Bernstein. ChaCha, a Variant of Salsa20. *Workshop Record of SASC 2008: The State of the Art of Stream Ciphers*, 2008. http://cr.yp.to/chacha.html.

[41] D. J. Bernstein. The Salsa20 Family of Stream Ciphers. *New Stream Cipher Designs*. Volume **4986**, Lecture Notes in Computer Science. Springer, 2008.

[42] D. J. Bernstein and P. Schwabe. NEON Crypto. *Cryptographic Hardware and Embedded Systems — CHES 2012*. Volume **7428**, Lecture Notes in Computer Science. Springer, 2012.

[43] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. On the Security of Keyed Sponge Constructions. Presented at SKEW 2011, 16-17 February 2011, Lyngby, Denmark, http://sponge.noekeon.org/SpongeKeyed.pdf.

[44] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. Permutation-based Encryption, Authentication and Authenticated Encryption. Presented at DIAC 2012, 05-06 July 2012, Stockholm, Sweden.

[45] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. Cryptographic Sponge Functions. 2008. http://sponge.noekeon.org/CSF-0.1.pdf.

[46] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. On the Indifferentiability of the Sponge Construction. *Advances in Cryptology — EUROCRYPT 2008*. Volume **4965**, Lecture Notes in Computer Science. Springer, 2008.

[47] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. The Keccak Sponge Function Family. 2008. http://keccak.noekeon.org.

[48] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. Duplexing the Sponge: Single-Pass Authenticated Encryption and Other Applications. *Selected Areas in Cryptography — SAC 2011*. Volume **7118**, Lecture Notes in Computer Science. Springer, 2011.

[49] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. On Alignment in Keccak. 2011. http://keccak.noekeon.org/KeccakAlignment.pdf.

[50] G. Bertoni, J. Daemen, M. Peeters, G. V. Assche, and R. v. Keer. Keccak Implementation Overview. 2012. http://keccak.noekeon.org.

[51] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. Sponge-Based Pseudo-Random Number Generators. *Cryptographic Hardware and Embedded Systems — CHES 2010*. Volume **6225**, Lecture Notes in Computer Science. Springer, 2010.

[52] A. Biere. Lingeling, Plingeling and Treengeling. `http://fmv.jku.at/lingeling/`.

[53] E. Biham. New Types of Cryptanalytic Attacks using Related Keys. *Journal of Cryptology*, **7**(4), 1994.

[54] E. Biham, A. Biryukov, and A. Shamir. Cryptanalysis of Skipjack Reduced to 31 Rounds Using Impossible Differentials. *Advances in Cryptology — EUROCRYPT 1999*. Volume **1592**, Lecture Notes in Computer Science. Springer, 1999.

[55] E. Biham, A. Biryukov, and A. Shamir. Miss in the Middle Attacks on IDEA and Khufu. *Fast Software Encryption — FSE 1999*. Volume **1636**, Lecture Notes in Computer Science. Springer, 1999.

[56] E. Biham and R. Chen. Near-Collisions of SHA-0. *Advances in Cryptology — CRYPTO 2004*. Volume **3152**, Lecture Notes in Computer Science. Springer, 2004.

[57] E. Biham, R. Chen, A. Joux, P. Carribault, C. Lemuet, and W. Jalby. Collisions of SHA-0 and Reduced SHA-1. *Advances in Cryptology — EUROCRYPT 2005*. Volume **3494**, Lecture Notes in Computer Science. Springer, 2005.

[58] E. Biham and A. Shamir. Differential Cryptanalysis of DES-like Cryptosystems. *Advances in Cryptology — CRYPTO 1990*. Volume **537**, Lecture Notes in Computer Science. Springer, 1991.

[59] E. Biham and A. Shamir. Differential Cryptanalysis of the Data Encryption Standard. Springer, 1993.

[60] E. Biham and A. Shamir. Differential Fault Analysis of Secret Key Cryptosystems. *Advances in Cryptology — CRYPTO 1997*. Volume **1294**, Lecture Notes in Computer Science. Springer, 1997.

[61] B. Bilgin, A. Bogdanov, M. Knežević, F. Mendel, and Q. Wang. FIDES: Lightweight Authenticated Cipher with Side-Channel Resistance for Constrained Hardware. *Cryptographic Hardware and Embedded Systems — CHES 2013*. Volume **8086**, Lecture Notes in Computer Science. Springer, 2013.

[62] A. Biryukov. DES-X (or DESX). *Encyclopedia of Cryptography and Security (2nd Ed.)* Springer, 2011.

[63] A. Biryukov and D. Khovratovich. PPAE: Parallelizable Permutation-based Authenticated Encryption. Presented at DIAC 2013, 11-13 August 2013, Chicago, USA, `http://2013.diac.cr.yp.to/slides/khovratovich.pdf`.

[64] A. Biryukov, A. Roy, and V. Velichkov. Differential Analysis of Block Ciphers SIMON and SPECK. Cryptology ePrint Archive, Report 2014/922. 2014. `http://eprint.iacr.org/2014/922`.

[65] A. Biryukov and D. Wagner. Slide Attacks. *Fast Software Encryption — FSE 1999*. Volume **1636**, Lecture Notes in Computer Science. Springer, 1999.

[66] A. Biryukov and D. Wagner. Advanced Slide Attacks. *Advances in Cryptology — EUROCRYPT 2000*. Volume **1807**, Lecture Notes in Computer Science. Springer, 2000.

[67] J. Black. Authenticated Encryption. *Encyclopedia of Cryptography and Security*. Springer, 2005.

[68] J. Blömer, M. Otto, and J.-P. Seifert. Sign Change Fault Attacks on Elliptic Curve Cryptosystems. *Fault Diagnosis and Tolerance in Cryptography — FDTC 2006*. Volume **4236**, Lecture Notes in Computer Science. Springer, 2006.

[69] M. Boesgaard, M. Vesterager, and E. Zenner. The Rabbit Stream Cipher. *New Stream Cipher Designs*. Volume **4986**, Lecture Notes in Computer Science. Springer, 2008.

[70] A. Bogdanov, I. Kizhvatov, and A. Pyshkin. Algebraic Methods in Side-Channel Collision Attacks and Practical Collision Detection. *Progress in Cryptology — INDOCRYPT 2008*. Volume **5365**, Lecture Notes in Computer Science. Springer, 2008.

[71] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. Robshaw, Y. Seurin, and C. Vikkelsoe. PRESENT: An Ultra-Lightweight Block Cipher. *Cryptographic Hardware and Embedded Systems — CHES 2007*. Volume **4727**, Lecture Notes in Computer Science. Springer, 2007.

[72] A. Bogdanov, M. Kňezević, G. Leander, D. Toz, K. Varici, and I. Verbauwhede. SPONGENT: A Lightweight Hash Function. *Cryptographic Hardware and Embedded Systems — CHES 2011*. Volume **6917**, Lecture Notes in Computer Science. Springer, 2011.

[73] A. Bogdanov, F. Mendel, F. Regazzoni, V. Rijmen, and E. Tischhauser. ALE: AES-based Lightweight Authenticated Encryption. *Fast Software Encryption — FSE 2013*. Volume **8424**, Lecture Notes in Computer Science. Springer, 2014.

[74] A. Bogdanov and V. Rijmen. Linear Hulls with Correlation Zero and Linear Cryptanalysis of Block Ciphers. Cryptology ePrint Archive, Report 2011/123. 2011. `http://eprint.iacr.org/2011/123`.

[75] A. Bogdanov and E. Tischhauser. On the Wrong Key Randomisation and Key Equivalence Hypotheses in Matsui's Algorithm 2. *Fast Software Encryption — FSE 2013*. Volume **8424**, Lecture Notes in Computer Science. Springer, 2014.

[76] A. Bogdanov and M. Wang. Zero Correlation Linear Cryptanalysis with Reduced Data Complexity. *Fast software encryption — fse 2012*. Volume **7549**, Lecture Notes in Computer Science. Springer, 2012.

[77] D. Boneh, R. A. DeMillo, and R. J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults. *Advances in Cryptology — EUROCRYPT 1997*. Volume **1233**, Lecture Notes in Computer Science. Springer, 1997.

[78] J. Borghoff, A. Canteaut, T. Güneysu, E. B. Kavun, M. Kňezević, L. R. Knudsen, G. Leander, V. Nikov, C. Paar, C. Rechberger, P. Rombouts, S. r. S. Thomsen, and T. Yalçin. PRINCE – A Low-Latency Block Cipher for Pervasive Computing Applications. *Advances in Cryptology — ASIACRYPT 2012*. Volume **7658**, Lecture Notes in Computer Science. Springer, 2012.

[79] L. Breveglieri, I. Koren., and P. Maistri. A Fault Attack Against the FOX Cipher Family. *Fault Diagnosis and Tolerance in Cryptography — FDTC 2006*. Volume **4236**, Lecture Notes in Computer Science, 2006.

[80] B. B. Brumley and N. Tuveri. Remote Timing Attacks are Still Practical. *European Symposium on Research in Computer Security — ESORICS 2011*. Volume **6879**, Lecture Notes in Computer Science. Springer, 2011.

[81] D. Brumley and D. Boneh. Remote Timing Attacks Are Practical. *Proceedings of the 12th Conference on USENIX Security Symposium — Volume 12*. SSYM'03. USENIX Association, 2003.

[82] R. Brummayer and A. Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. *Tools and Algorithms for the Construction and Analysis of Systems*. Volume **5505**, Lecture Notes in Computer Science. Springer, 2009. `http://fmv.jku.at/boolector/`.

[83] B. Buchberger. Ein Algorithmus zum Auffinden der Basiselemente des Restklassenrings nach einem Nulldimensionalen Polynomideal. PhD thesis. 1965.

[84] CAESAR – Competition for Authenticated Encryption: Security, Applicability, and Robustness. `http://competitions.cr.yp.to/caesar.html`. 2014.

[85] C. D. Cannière, O. Dunkelman, and M. Kňezević. KATAN and KTANTAN: A Family of Small and Efficient Hardware-Oriented Block Ciphers. *Cryptographic Hardware and Embedded Systems — CHES 2009*. Volume **5747**, Lecture Notes in Computer Science. Springer, 2009.

[86] C. D. Cannière and B. Preneel. Trivium. *New Stream Cipher Designs*. Volume **4986**, Lecture Notes in Computer Science. Springer, 2008.

[87] C. Clavier, B. Gierlichs, and I. Verbauwhede. Fault Analysis Study of IDEA. *Topics in Cryptology — CT-RSA 2008*. Volume **4964**, Lecture Notes in Computer Science, 2008.

[88] D. Coppersmith. The Data Encryption Standard (DES) and its Strength Against Attacks. *IBM Journal of Research and Development*, **38**(3), 1994.

[89] N. T. Courtois. Algebraic Attacks on Combiners with Memory and Several Outputs. *International Conference on Information Security and Cryptology — ICISC 2004*. Volume **3506**, Lecture Notes in Computer Science. Springer, 2004. http://eprint.iacr.org/2003/125.

[90] N. T. Courtois, D. Hulme, and T. Mourouzis. Solving Circuit Optimisation Problems in Cryptography and Cryptanalysis. *Special-purpose Hardware for Attacking Cryptographic Systems — SHARCS 2012*, 2012. http://eprint.iacr.org/2011/475.

[91] N. T. Courtois, K. Nohl, and S. O'Neil. Algebraic Attacks on the Crypto-1 Stream Cipher in MiFare Classic and Oyster Cards. Cryptology ePrint Archive, Report 2008/166. 2008. http://eprint.iacr.org/2008/166.

[92] Cryptographic Competitions. http://competitions.cr.yp.to/.

[93] D. Câmara, C. P. Gouvêa, J. López, and R. Dahab. Fast Software Polynomial Multiplication on ARM Processors Using the NEON Engine. *Security Engineering and Intelligence Informatics*. Volume **8128**, Lecture Notes in Computer Science. Springer, 2013.

[94] J. Daemen, R. Govaerts, and J. Vandewalle. Correlation Matrices. *Fast Software Encryption — FSE 1994*. Volume **1008**, Lecture Notes in Computer Science. Springer, 1995.

[95] J. Daemen and V. Rijmen. AES Proposal: Rjindael. 1998. http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf.

[96] J. Daemen and V. Rijmen. The Advanced Encryption Standard. 2001. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[97] J. Daemen and V. Rijmen. The Design of Rijndael. Springer, 2002.

[98] Data Encryption and Integrity Algorithms. Preliminary State Standard of Republic of Belarus (STB P 34.101.31–2007), http://apmi.bsu.by/assets/files/std/belt-spec27.pdf.

[99] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, **7**(3), 1960.

[100] L. De Moura and N. Bjørner. Satisfiability Modulo Theories: Introduction and Applications. *Communications of the ACM*, **54**(9), 2011.

[101] A. Dehbaoui, J.-M. Dutertre, B. Robisson, and A. Tria. Electromagnetic Transient Faults Injection on a Hardware and a Software Implementations of AES. *Fault Diagonsis and Tolerance in Cryptography — FDTC 2012*. IEEE, 2012.

[102] R. Denning and D. Elizabeth. Cryptography and Data Security. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1982.

[103] O. Derouet. Secure Smartcard Design Against Laser Fault Injection Attacks. Invited talk at Fault Diagnosis and Tolerance in Cryptography, 2007.

[104] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. Internet Requests for Comments 5246, `http://tools.ietf.org/html/rfc5246`. 2008.

[105] I. Dinur and A. Shamir. Cube Attacks on Tweakable Black Box Polynomials. *Advances in Cryptology — EUROCRYPT 2009*. Volume **5479**, Lecture Notes in Computer Science. Springer, 2009.

[106] D. Engels, M.-J. O. Saarinen, P. Schweitzer, and E. M. Smith. The Hummingbird-2 Lightweight Authenticated Encryption Algorithm. *International Conference on Security and Privacy — RFIDSec 2012*. Volume **7055**, Lecture Notes in Computer Science. Springer, 2012.

[107] eSTREAM - the ECRYPT Stream Cipher Project. `http://www.ecrypt.eu.org/stream`. 2004-2008.

[108] J.-C. Faugère. A New Efficient Algorithm for Computing Gröbner Bases (F4). *Journal of Pure and Applied Algebra*, **139**(1), 1999.

[109] J.-C. Faugère. A New Efficient Algorithm for Computing Gröbner Bases Without Reduction to Zero (F5). *International Symposium on Symbolic and Algebraic Computation — ISSAC 2002*. ACM, 2002.

[110] J.-C. Faugère, P. Gianni, D. Lazard, and T. Mora. Efficient Computation of Zero-dimensional Groebner Bases by Change of Ordering. *Journal of Symbolic Computation*, **16**(4), 1993.

[111] N. Ferguson. Collision Attacks on OCB. Tech. rep. 2002. `http://web.cs.ucdavis.edu/~rogaway/ocb/fe02.pdf`.

[112] N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas, and J. Walker. The Skein Hash Function Family. 2010. `http://www.skein-hash.info/sites/default/files/skein1.3.pdf`.

[113] N. Ferguson, D. Whiting, B. Schneier, J. Kelsey, S. Lucks, and T. Kohno. Helix – Fast Encryption and Authentication in a Single Cryptographic Primitive. 2003. `https://www.schneier.com/paper-helix.html`.

[114] S. Fluhrer, I. Mantin, and A. Shamir. Weaknesses in the Key Scheduling Algorithm of RC4. *Selected Areas in Cryptography — SAC 2001*. Volume **2259**, Lecture Notes in Computer Science. Springer, 2001.

[115] V. Ganesh. Decision Procedures for Bit-Vectors, Arrays and Integers. PhD thesis. 2007.

[116] V. Ganesh, R. Govostes, K. Y. Phang, M. Soos, and E. Schwartz. STP — A Simple Theorem Prover, 2006-2014. `http://stp.github.io/stp`.

[117] M. Garey and D. Johnson. Computers and Intractability – A Guide to the Theory of NP-Completeness. W.H. Freeman and Company, 1979.

[118] P. Gauravaram, L. R. Knudsen, K. Matusiewicz, F. Mendel, C. Rechberger, M. Schläffer, and S. S. Thomsen. Grøstl – a SHA-3 Candidate. 2011. `http://www.groestl.info/`.

[119] M. Gorski, S. Lucks, and T. Peyrin. Slide Attacks on a Class of Hash Functions. *Advances in Cryptology — ASIACRYPT 2008*. Volume **5350**, Lecture Notes in Computer Science. Springer, 2008.

[120] L. K. Grover. A Fast Quantum Mechanical Algorithm for Database Search. *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*. STOC '96. ACM, 1996.

[121] S. Gueron. AES-GCM Software Performance on the Current High End CPUs as a Performance Baseline for CAESAR Competition. Presented at DIAC 2013, 11-13 August 2013, Chicago, USA, `http://2013.diac.cr.yp.to/slides/gueron.pdf`.

[122] S. Gueron. Intel Advanced Encryption Standard (AES) Instructions Set. `https://software.intel.com/en-us/articles/intel-advanced-encryption-standard-aes-instructions-set/`.

[123] S. Gueron. Intel Carry-Less Multiplication Instruction and its Usage for Computing the GCM Mode. `https://software.intel.com/en-us/articles/intel-carry-less-multiplication-instruction-and-its-usage-for-computing-the-gcm-mode`.

[124] J. Guo, P. Karpman, I. Nikolić, L. Wang, and S. Wu. Analysis of BLAKE2. *Topics in Cryptology — CT-RSA 2014*. Volume **8366**, Lecture Notes in Computer Science. Springer, 2014.

[125] J. Guo, T. Peyrin, and A. Poschmann. The PHOTON Family of Lightweight Hash Functions. *Advances in Cryptology — CRYPTO 2011*. Volume **6841**, Lecture Notes in Computer Science. Springer, 2011.

[126] J. Guo, T. Peyrin, A. Poschmann, and M. Robshaw. The LED Block Cipher. *Cryptographic Hardware and Embedded Systems — CHES 2011*. Volume **6917**, Lecture Notes in Computer Science. Springer, 2011.

[127] P. Gutmann. Encrypt-then-MAC for Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). Internet Requests for Comments 7366, `http://tools.ietf.org/html/rfc7366`. 2014.

[128] M. Hellman. A Cryptanalytic Time-Memory Trade-Off. *Information Theory, IEEE Transactions on*, **26**(4), 1980.

[129] L. Henzen, F. Carbognani, N. Felber, and W. Fichtner. VLSI Hardware Evaluation of the Stream Ciphers Salsa20 and ChaCha, and the Compression Function Rumba. *International Conference on Signals, Circuits and Systems — SCS 2008*. IEEE, 2008.

[130] M. Hojsík and B. Rudolf. Differential Fault Analysis of Trivium. *Fast Software Encryption — FSE 2008*. Volume **5086**, Lecture Notes in Computer Science. Springer, 2008.

[131] M. Hojsík and B. Rudolf. Floating Fault Analysis of Trivium. *Progress in Cryptology — INDOCRYPT 2008*. Volume **5365**, Lecture Notes in Computer Science. Springer, 2008.

[132] IBM. MARS. 1998.

[133] IEEE. 802.1ae – Media Access Control (MAC) Securityl. 2006.

[134] E. S. Inc. CLP-15: Ultra-High Throughput AES-GCM Core-40 Gbps. 2008.

[135] Intel ® Architecture Instruction Set Extensions Programming Reference. Intel Corporation, 2014. `http://software.intel.com/en-us/intel-isa-extensions`.

[136] A. Joux. Authentication Failures in NIST Version of GCM. 2006. `http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/Joux_comments.pdf`.

[137] P. Jovanovic and M. Kreuzer. Algebraic Attacks using SAT-Solvers. *Groups — Complexity — Cryptology*, **2**, 2010.

[138] P. Jovanovic, M. Kreuzer, and I. Polian. An Algebraic Fault Attack on the LED Block Cipher. Third International Conference on Symbolic Computation and Cryptography — SCC 2012.

[139] P. Jovanovic, M. Kreuzer, and I. Polian. Multi-Stage Fault Attacks on Block Ciphers. 14th Workshop on RTL and High Level Testing — WRTLT 2013.

[140] P. Jovanovic, M. Kreuzer, and I. Polian. A Fault Attack on the LED Block Cipher. *International Workshop on Constructive Side-Channel Analysis and Secure Design — COSADE 2012*. Volume **7275**, Lecture Notes in Computer Science. Springer, 2012.

[141] P. Jovanovic, A. Luykx, and B. Mennink. Beyond $2^{c/2}$ Security in Sponge-Based Authenticated Encryption Modes. *Advances in Cryptology — ASIACRYPT 2014*. Volume **8873**, Lecture Notes in Computer Science. Springer, 2014.

[142] P. Jovanovic and S. Neves. Practical Cryptanalysis of the Open Smart Grid Protocol. *Fast Software Encryption — FSE 2015*. Springer, 2015. (to appear).

[143] P. Jovanovic and I. Polian. Fault-based Attacks on the Bel-T Block Cipher Family. *Design, Automation and Test in Europe — DATE 2015*. EDA Consortium, 2015.

[144] P. Junod and S. Vaudenay. FOX: A New Family of Block Ciphers. *Selected Areas in Cryptography — SAC 2005*. Volume **3357**, Lecture Notes in Computer Science. Springer, 2005.

[145] P. Junod and S. Vaudenay. Perfect Diffusion Primitives for Block Ciphers. *Selected Areas in Cryptography — SAC 2005*. Volume **3357**, Lecture Notes in Computer Science. Springer, 2005.

[146] D. Kahn. The Codebreakers: The Story of Secret Writing. Macmillan Press, 1967.

[147] B. Kaliski. PKCS #5: Password-Based Cryptography Specification Version 2.0. Internet Requests for Comments 2898, `https://tools.ietf.org/html/rfc2898`. 2000.

[148] J. Kaliski B. S. and M. Robshaw. Linear Cryptanalysis Using Multiple Approximations. *Advances in Cryptology — CRYPTO 1994*. Volume **839**, Lecture Notes in Computer Science. Springer, 1994.

[149] E. Käsper and P. Schwabe. Faster and Timing-Attack Resistant AES-GCM. IACR Cryptology ePrint Archive, Report 2009/129. 2009. `http://eprint.iacr.org/2009/129`.

[150] S. Kent and K. Seo. Security Architecture for the Internet Protocol. Internet Requests for Comments 4301, `http://tools.ietf.org/html/rfc4301`. 2005.

[151] A. Kerckhoffs. La Cryptographie Militaire. *Journal des Sciences Militaires*, **IX**, 1883.

[152] T. Keresztfalvi and M. Salomon. CronorX — A NORX ASIC Implementation. 2014. Supervised by F. K. Gürkaynak and C. Keller, `http://iis-projects.ee.ethz.ch/index.php/NORX_-_an_AEAD_algorithm_for_the_CAESAR_competition`.

[153] D. Khovratovich and I. Nikolić. Rotational Cryptanalysis of ARX. *Fast Software Encryption — FSE 2010*. Volume **6147**, Lecture Notes in Computer Science. Springer, 2010.

[154] D. Khovratovich, I. Nikolić, and C. Rechberger. Rotational Rebound Attacks on Reduced Skein. *Advances in Cryptology — ASIACRYPT 2010*. Volume **6477**, Lecture Notes in Computer Science. Springer, 2010.

[155] J. Killian and P. Rogaway. How to Protect DES Against Exhaustive Key Search. *Advances in Cryptology — CRYPTO 1996*. Volume **1109**, Lecture Notes in Computer Science. Springer, 1996.

[156] C. Kim and J.-J. Quisquater. Fault Attacks for CRT Based RSA: New Attacks, New Results, and New Countermeasures. *Workshop on Information Security, Theory and Practices — WISTP 2007*. Volume **4462**, Lecture Notes in Computer Science. Springer, 2007.

[157] A. Klein. Attacks on the RC4 Stream Cipher. *Designs, Codes and Cryptography*, **48**(3), 2008.

[158] M. Knežević, V. Nikov, and P. Rombouts. Low-Latency Encryption – Is "Lightweight = Light + Wait"? *Cryptographic Hardware and Embedded Systems — CHES 2012*. Volume **7428**, Lecture Notes in Computer Science. Springer, 2012.

[159] L. R. Knudsen. Truncated and Higher Order Differentials. *Fast Software Encryption — FSE 1995*. Volume **1008**, Lecture Notes in Computer Science. Springer, 1995.

[160] L. R. Knudsen. DEAL – A 128-bit Block Cipher. *NIST AES Proposal*, 1998.

[161] L. R. Knudsen and J. E. Mathiassen. A Chosen-Plaintext Linear Attack on DES. *Fast Software Encryption — FSE 2001*. Volume **1978**, Lecture Notes in Computer Science. Springer, 2001.

[162] L. R. Knudsen and M. J. B. Robshaw. The Block Cipher Companion. Springer, 2011.

[163] L. R. Knudsen and M. Robshaw. Non-Linear Approximations in Linear Cryptanalysis. *Advances in Cryptology — EUROCRYPT 1996*. Volume **1070**, Lecture Notes in Computer Science. Springer, 1996.

[164] D. E. Knuth. The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1. Vol. 4A. Addison-Wesley, 2011. `http://www-cs-faculty.stanford.edu/~uno/taocp.html`.

[165] Ç. K. Koç. Cryptographic Engineering. Springer, 2009.

[166] P. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS and Other Systems. *Advances in Cryptology — CRYPTO 1996*. Volume **1109**, Lecture Notes in Computer Science. Springer, 1996.

[167] P. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. *Advances in Cryptology — CRYPTO 1999*. Volume **1666**, Lecture Notes in Computer Science. Springer, 1999.

[168] I. Koren and C. M. Krishna. Fault-tolerant Systems. Morgan Kaufmann Publishers Inc., 2007.

[169] M. Kreuzer and L. Robbiano. Computational Commutative Algebra 1. Springer, 2000.

[170] M. Kreuzer and L. Robbiano. Computational Commutative Algebra 2. Springer, 2005.

[171] T. Krovetz and P. Rogaway. The Software Performance of Authenticated-Encryption Modes. *Fast Software Encryption — FSE 2011*. Volume **6733**, Springer, 2011.

[172] R. Kumar, P. Jovanovic, W. Burleson, and I. Polian. Parametric Trojans for Fault-Injection Attacks on Cryptographic Hardware. *Fault Diagnosis and Tolerance in Cryptography — FDTC 2014*. IEEE, 2014.

[173] R. Kumar, P. Jovanovic, and I. Polian. Precise Fault-Injections using Voltage and Temperature Manipulation for Differential Cryptanalysis. *20th International On-Line Testing Symposium — IOLTS 2014*. IEEE, 2014.

[174] K. Kursawe and C. Peters. Structural Weaknesses in the Open Smart Grid Protocol. Cryptology ePrint Archive, Report 2015/088. 2015. `http://eprint.iacr.org/2015/088`.

[175] W. Ladd. McMambo. CAESAR Proposal. 2014. `http://competitions.cr.yp.to/round1/mcmambov1.pdf`.

[176] X. Lai and J. L. Massey. A Proposal for a New Block Encryption Standard. *Advances in Cryptology — EUROCRYPT 1990*. Volume **473**, Lecture Notes in Computer Science. Springer, 1991.

[177] X. Lai, J. L. Massey, and S. Murphy. Markov Ciphers and Differential Cryptanalysis. *Advances in Cryptology — EUROCRYPT 1991*. Volume **547**, Lecture Notes in Computer Science. Springer, 1991.

[178] H. Lipmaa and S. Moriai. Efficient Algorithms for Computing Differential Properties of Addition. *Fast Software Encryption — FSE 2001*. Volume **2355**, Lecture Notes in Computer Science. Springer, 2001.

[179] A. Ltd. GCM Extension for AES G3 Core. 2007.

[180] J. Marques-Silva and K. Sakallah. GRASP: a Search Algorithm for Propositional Satisfiability. *Computers, IEEE Transactions on*, **48**(5), 1999.

[181] Mate Soos. CryptoMinisat. `http://www.msoos.org/cryptominisat4`.

[182] M. Matsui. Linear Cryptanalysis Method for DES. *Advances in Cryptology — EUROCRYPT 1993*. Volume **765**, Lecture Notes in Computer Science. Springer, 1994.

[183] M. Matsui and A. Yamagishi. A New Method for Known Plaintext Attack of FEAL Cipher. *Advances in Cryptology — EUROCRYPT 1992*. Volume **658**, Lecture Notes in Computer Science. Springer, 1993.

[184] P. Maxwell. Wheesht. CAESAR Proposal. 2014. `http://competitions.cr.yp.to/round1/wheeshtv03.pdf`.

[185] K. Minematsu, S. Lucks, H. Morita, and T. Iwata. Attacks and Security Proofs of EAX-Prime. Cryptology ePrint Archive, Report 2012/018. 2012. `http://eprint.iacr.org/2012/018`.

[186] P. Morawiecki, J. Pieprzyk, and M. Srebrny. Rotational Cryptanalysis of Round-Reduced Keccak. IACR Cryptology ePrint Archive, Report 2012/546. 2012. http://eprint.iacr.org/2012/546.

[187] N. Mouha and B. Preneel. Towards Finding Optimal Differential Characteristics for ARX: Application to Salsa20. IACR Cryptology ePrint Archive, Report 2013/328. 2013. http://eprint.iacr.org/2013/328.

[188] L. d. Moura and N. Bjørner. Z3: An Efficient SMT Solver. *Tools and Algorithms for the Construction and Analysis of Systems*. Volume **4963**, Lecture Notes in Computer Science. Springer, 2008.

[189] M. Mozaffari-Kermani and A. Reyhani-Masoleh. Efficient and High-Performance Parallel Hardware Architectures for the AES-GCM. *Computers, IEEE Transactions on*, **61**(8), 2012.

[190] M. Muehlberghuber, C. Keller, F. K. Gürkaynak, and N. Felber. FPGA-Based High-Speed Authenticated Encryption System. English. *VLSI-SoC: From Algorithms to Circuits and System-on-Chip Design*. Volume **418**, IFIP Advances in Information and Communication Technology. Springer, 2013.

[191] D. Mukhopadhyay. An Improved Fault Based Attack of the Advanced Encryption Standard. *Progress in Cryptology — AFRICACRYPT 2009*. Volume **5580**, Lecture Notes in Computer Science. Springer, 2009.

[192] F. Muller. Differential Attacks against the Helix Stream Cipher. *Fast Software Encryption — FSE 2004*. Volume **3017**, Lecture Notes in Computer Science. Springer, 2004.

[193] S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. 2008. https://bitcoin.org/bitcoin.pdf.

[194] National Institute of Standards and Technology. DES Modes of Operation, FIPS Publication 81. 1980.

[195] National Institute of Standards and Technology. Recommendation for Block Cipher Modes of Operation: Methods and Techniques. 2001. http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf.

[196] National Institute of Standards and Technology. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. 2007. http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf.

[197] M. A. Nielsen and I. L. Chuang. Quantum Computation and Quantum Information. Cambridge University Press, 2011.

[198] Niklas Eén and Niklas Sörensson. MiniSat. http://minisat.se/.

[199] NODE — The NORX Differential Search Engine. 2014. https://github.com/norx/NODE.

[200] P. Oechslin. Making a Faster Cryptanalytic Time-Memory Trade-Off. *Advances in Cryptology — CRYPTO 2003*. Volume **2729**, Lecture Notes in Computer Science. Springer, 2003.

[201] M.-J. O.Saarinen. Related-Key Attacks Against Full Hummingbird-2. *Fast Software Encryption — FSE 2013*. Lecture Notes in Computer Science. Springer, 2014.

[202] C. Paar and J. Pelzl. Understanding Cryptography. Springer, 2010.

[203] S. Paul and B. Preneel. Solving Systems of Differential Equations of Addition. *Australasian Conference on Information Security and Privacy — ACISP 2005*. Volume **3574**, Lecture Notes in Computer Science. Springer, 2005.

[204] C. Percival. Stronger Key Derivation via Sequential Memory-Hard Functions. `https://www.tarsnap.com/scrypt/scrypt.pdf`. 2009.

[205] T. Peyrin. Security Analysis of Extended Sponge Functions. Presented at the ECRYPT Workshop Hash Functions in Cryptology: Theory and Practice, Leiden, The Netherlands, June 4th 2008, `http://www.lorentzcenter.nl/lc/web/2008/309/presentations/Peyrin.pdf`.

[206] PHC – Password Hashing Competition. `https://password-hashing.net`. 2014.

[207] A. Popov. Prohibiting RC4 Cipher Suites. Internet Requests for Comments 7465, `http://tools.ietf.org/html/rfc7465`. 2015.

[208] D. Priemuth-Schmid and A. Biryukov. Slid Pairs in Salsa20 and Trivium. *Progress in Cryptology — INDOCRYPT 2008*. Volume **5365**, Lecture Notes in Computer Science. Springer, 2008. `http://eprint.iacr.org/2008/405`.

[209] N. Provos and D. Mazières. A Future-adaptive Password Scheme. *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '99. USENIX Association, 1999.

[210] R. L. Rivest, M. Robshaw, R. Sidney, and Y. L. Yin. The RC6 Block Cipher. 1998. `http://people.csail.mit.edu/rivest/pubs/RRSY98.pdf`.

[211] P. Rogaway. Authenticated-Encryption with Associated-Data. *ACM Conference on Computer and Communications Security — CCS 2002*. ACM press, 2002.

[212] P. Rogaway. Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC. *Advances in Cryptology — ASIACRYPT 2004*. Volume **3329**, Lecture Notes in Computer Science. Springer, 2004.

[213] P. Rogaway. Nonce-based Symmetric Encryption. *Fast Software Encryption — FSE 2004*. Volume **3017**, Lecture Notes in Computer Science. Springer, 2004.

[214] P. Rogaway, M. Bellare, J. Black, and T. Krovetz. OCB: A Block-Cipher Mode of Operation for Efficient Authenticated Encryption. *ACM Conference on Computer and Communications Security — CCS 2001*. ACM, 2001.

[215] P. Rogaway and D. Wagner. A Critique of CCM. 2003. http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/comments/800-38_Series-Drafts/CCM/RW_CCM_comments.pdf.

[216] D. Saha, D. Mukhopadhyay, and D. RoyChowdhury. A Diagonal Fault Attack on the Advanced Encryption Standard. IACR Cryptology ePrint Archive, Report 2009/581. 2009. http://eprint.iacr.org/2009/581.

[217] SAT Competitions. http://www.satcompetition.org.

[218] Satisfiability Modulo Theories Competition. http://www.smtcomp.org/.

[219] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Ferguson. Twofish: A 128-Bit Block Cipher. 1998. https://www.schneier.com/paper-twofish-paper.pdf.

[220] A. A. Selçuk. On Probability of Success in Linear and Differential Cryptanalysis. *Journal of Cryptology*, **21**(1), 2008.

[221] P. Sepehrdad, S. Vaudenay, and M. Vuagnoux. Discovery and Exploitation of New Biases in RC4. *Selected Areas in Cryptography — SAC 2011*. Volume **6544**, Lecture Notes in Computer Science. Springer, 2011.

[222] SHA-3 Competition. http://csrc.nist.gov/groups/ST/hash/sha-3/Round3/index.html. 2007-2012.

[223] C. E. Shannon. A Mathematical Theory of Communication. *Bell Systems Technical Journal*, **27**, 1948.

[224] C. E. Shannon. Communication Theory of Secrecy Systems. *Bell Systems Technical Journal*, **28**(4), 1949.

[225] Z. Shi, B. Zhang, D. Feng, and W. Wu. Improved Key Recovery Attacks on Reduced Round Salsa20 and ChaCha. *International Conference on Information Security and Cryptology — ICISC 2012*. Volume **7839**, Lecture Notes in Computer Science. Springer, 2012.

[226] P. W. Shor. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM Journal of Computing*, **26**(5), 1997.

[227] V. Shoup. A Computational Introduction to Number Theory and Algebra. Cambridge University Press, 2nd ed., 2009. http://shoup.net/ntb.

[228] L. Song and L. Hu. Differential Fault Attack on the PRINCE Block Cipher. IACR Cryptology ePrint Archive, Report 2013/043. 2013. http://eprint.iacr.org/2013/043.

[229] W. Stein. Sage Mathematics Software. The Sage Development Team, 2005-2015. http://sagemath.org.

[230] SUPERCOP – System for Unified Performance Evaluation Related to Cryptographic Operations and Primitives. `http://bench.cr.yp.to/supercop.html`.

[231] H. Technology. AES-GCM Cores. 2007.

[232] E. Tews, R.-P. Weinmann, and A. Pyshkin. Breaking 104 Bit WEP in Less Than 60 Seconds. *Information Security Applications*. Volume **4867**, Lecture Notes in Computer Science. Springer, 2007.

[233] Y. Tsunoo, T. Saito, H. Kubo, T. Suzaki, and H. Nakashima. Differential Cryptanalysis of Salsa20/8. *The State of the Art of Stream Ciphers (SASC)*, 2007.

[234] Y. Tsunoo, T. Saito, T. Suzaki, M. Shigeri, and H. Miyauchi. Cryptanalysis of DES Implemented on Computers with Cache. *Cryptographic Hardware and Embedded Systems — CHES 2003*. Volume **2779**, Lecture Notes in Computer Science. Springer, 2003.

[235] M. Tunstall, D. Mukhopadhyay, and S. Ali. Differential Fault Analysis of the Advanced Encryption Standard Using a Single Fault. *Workshop on Information Security, Theory and Practices — WISTP 2011*. Volume **6633**, Lecture Notes in Computer Science. Springer, 2011.

[236] S. Vaudenay. On the Lai-Massey Scheme. *Advances in Cryptology — ASIACRYPT 1999*. Volume **1716**, Lecture Notes in Computer Science. Springer, 1999.

[237] S. Vaudenay. Security Flaws Induced by CBC Padding – Applications to SSL, IPSEC, WTLS... *Advances in Cryptology — EUROCRYPT 2002*. Volume **2332**, Lecture Notes in Computer Science. Springer, 2002.

[238] D. Wagner. The Boomerang Attack. *Fast Software Encryption — FSE 1999*. Volume **1636**, Lecture Notes in Computer Science. Springer, 1999.

[239] X. Wang, Y. L. Yin, and H. Yu. Efficient Collision Search Attacks on SHA-0. *Advances in Cryptology — CRYPTO 2005*. Volume **3621**, Lecture Notes in Computer Science. Springer, 2005.

[240] X. Wang, Y. L. Yin, and H. Yu. Finding Collisions in the Full SHA-1. *Advances in Cryptology — CRYPTO 2005*. Volume **3621**, Lecture Notes in Computer Science. Springer, 2005.

[241] X. Wang and H. Yu. How to Break MD5 and Other Hash Functions. *Advances in Cryptology — EUROCRYPT 2005*. Volume **3494**, Lecture Notes in Computer Science. Springer, 2005.

[242] D. Whiting, R. Housley, and N. Ferguson. Counter with CBC-MAC (CCM). Internet Requests for Comments 3610, `http://tools.ietf.org/html/rfc3610`. 2003.

[243] D. Whiting, B. Schneier, J. Kelsey, S. Lucks, and F. Muller. Phelix – Fast Encryption and Authentication in a Single Cryptographic Primitive. ECRYPT Stream Cipher Project Report 2005/027, `https://www.schneier.com/paper-phelix.html`. 2005.

[244] M. J. Wiener. The Full Cost of Cryptanalytic Attacks. *Journal of Cryptology*, **17**(2), 2004.

[245] H. Wu. The Stream Cipher HC-128. *New Stream Cipher Designs*. Volume **4986**, Lecture Notes in Computer Science. Springer, 2008.

[246] H. Wu. The Hash Function JH. 2011. `http://www3.ntu.edu.sg/home/wuhj/research/jh/jh_round3.pdf`.

[247] H. Wu and B. Preneel. Differential-Linear Attacks Against the Stream Cipher Phelix. *Fast Software Encryption — FSE 2007*. Volume **4593**, Lecture Notes in Computer Science. Springer, 2007.

[248] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Transport Layer Protocol. Internet Requests for Comments 4253, `http://tools.ietf.org/html/rfc4253`. 2006.

Test Vectors for NORX

All of the following test vectors and intermediate values are denoted in hexadecimal little-endian format.

Traces for G and F

Let $x = (a, b, c, d)$ and $s = (s_0, \ldots, s_{15})$, where a, b, c, d, and s_i for $0 \leq i \leq 15$ are w-bit sized integers. Below we show the results after $n \in \{0, \ldots, 15\}$ iterations of 32- and 64-bit G and F for the input at $n = 0$.

n	$\mathsf{G}^n(x)$				$\mathsf{G}^n(x)$			
0	00000001	00000000	00000000	00000000	0000000000000001	0000000000000000	0000000000000000	0000000000000000
1	00002001	42024200	21010100	20010100	0000002000000001	4200004002020000	2100000001010000	2000000001010000
2	8E6E6E29	8783068E	0FAD8F6F	6D4A8F0C	82600C6C420A0C61	9CC80A07051612D8	0E04090FC283050C	6C22090F6183020C
3	9D6D8718	B987D564	C7474857	24F32605	CFE22B520239332F	1A1C7E83367612BF	7CD42AB6581A2C29	4D6E1FA833FA9021
4	C291D60D	19A0D5EA	F0E0D424	C473BBA3	3D5E2BDE88AFF93B	ECC8D95156DCEC38	0A0446A133EE8271	C0F5AC2CAED2D2BB
5	35F2578F	853DBE90	195D7490	149F61EC	C3C467FE58784143	3C97782DEEAC5790	44843BC64A41A7D8	B25B09F9540B8D94
6	AF557EEA	C53796FE	163271AC	4E4F5CFB	AC67C4758EDABE16	C321E8CE8496C5ED	2FBCC28DAA1D48B5	A3FCA8310FEB2BC4
7	DBA043BC	6C09B0BE	4D7D55A3	4425E08E	4CE9BEBB0DD254A6	4F2485BE02121466	C67650560B60EF00	BF0EDED6FE7B2547
8	44912C86	E215F517	D0614027	A094B0F2	C2C12A34D0D5FD94	1C191DA55351ADFA	4149F615635943F5	D8772462D909E5A8
9	88980FED	27B1DE05	79A35152	3DCCE71E	377F4E2380FD20F6	3F886AF0AE6DF23C	47611212B6E0C0B6	203B070E18D179E3
10	BD333CDB	C51A4ECA	6DA7579C	F9F141A0	BBEA67F79ED0A639	2348E7B04B8FE61F	C4A0C4BDD549E690	4002E63DD2FAACBB
11	9F85298E	F2ACF75A	D4CEBDA4	D5A4BA0E	5BD7374482BCD0AE	E465F7287EA1095B	4F9713BAFD0E5037	ED98AE740EF44907
12	FA2D9998	3BD8B744	AFE0677A	1CBE34B0	9D12266D61EF2E4B	3F82247049FFDACF	01DA8A38AC6DB80D	6D791887DB7BC086
13	72B826FD	30355A44	D6CE9703	7CC51E11	CF409F3F352535F6	0E03BB546CA18FF7	F1FD88CBBC37921F	6D6F5580D65BB572
14	C7053F6D	D35DA308	CD1616B8	170BE75B	A15E9AA4D3881354	F042F1E89CECFFD3	2C3DF0918FBD9D82	E3104F8E503EEEB7
15	29F7F20E	3256FD7D	068E6667	AB330772	F487F38AFF72BCE1	99081BA0BCA194B5	2B8AA44DCF5F3898	8F7C4CF6C564757A

n	$\mathsf{F}^n(s)$				$\mathsf{F}^n(s)$			
0	00000001	00000000	00000000	00000000	0000000000000001	0000000000000000	0000000000000000	0000000000000000
	00000000	00000000	00000000	00000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000
	00000000	00000000	00000000	00000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000
	00000000	00000000	00000000	00000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000
1	04004001	20200400	20042020	4A4A8A08	0000004000000401	2020000400000000	2000042000000020	42400888420A0840
	01880885	8A424A40	4A024A02	C24A0248	1008008580981891	8240004842020800	4800020A00420200	C200084042420048
	41212104	888C4C4A	41210520	05212101	4100000021210004	8844080A80440408	4120000421010000	0420010100210100
	05012000	20202004	884A4A08	40210500	0400010100200000	2000000020200004	8802080A40420208	4020000401010000
2	EFDB6055	4EB0C8FD	4D66BAD5	A5716F6F	9D802FD127A732A1	BFDC94FCF7EDB4F6	50E28C54A198AD0E	09FCDB8FCCC9DDA8
	3315BA06	B5E09122	44A18E71	51E36297	7ACEC81E5BAA6D25	10C9CBCF5BFEFC27	11A152F2C1A43FCA	6BA77CCFA2D9F407
	F137B870	3C7265F6	00C30D5B	295A09AA	0E03AD8E4F36AD96	B405D697E680A2BB	3651B1301374F05D	EC2A3CD28E701034
	B42B85E7	AC007723	742077A7	4BADCF9B	D793C96953AA22B3	81B56FC8F78827DD	A5F18C894182A861	F95F620C599E1A7D
3	B49E8FA1	B87AED22	86152D27	BEB398AD	6D9C774FB118B930	0AD4888256442919	B2625AFA68288616	3F682524B541B12D
	BD48EB80	1D4447DA	B7458BA9	A9E9EF9B	09FB30C77ED1253C	D276B00A56FA3BB2	D1A3ED2B432628E0	59DE47C408703466
	F7599C6A	203FB309	694A1283	C4875743	730C85F6CF7CD9B4	D731F331C620402D	664456562656A61E	10F001A72ABF1CCA
	F4E78B62	50BE8206	7BEF5DF7	F92F6B9C	E04F26164B84BCD5	E1CE43EA4AC71790	BE0A7BDA26AB8C3E	083CB972BE746F0D

n	$\mathsf{F}^n(s)$				$\mathsf{F}^n(s)$			
4	D8936EA9	4FDFA7F9	2E23D116	ED7C3692	9AE671BAC4106A33	2532A3AF80EB8C24	8807B8748AAF89BB	CCBD275D7AC0180C
	3E463C40	A5AA5D55	A05A6E11	D22C7D58	9E3C9A644E2EE2B1	6EF830BF37A17BB2	A56A3F09DA96ABC9	6674A590854EA97D
	3C0D461D	5D78E74F	88C9121B	ECA4CA13	D58BFB1A8D2677C5	5696D8DEA26A6D6D	2E973803C96922A4	9C8EC44641A390FD
	E12928CB	0167E06D	90E1494E	7CBBCCDA	ABE2F120F069F77A	305FE9E02B725884	1D2A9380316FE1A6	8FA5B15C10F77415
5	DC4D4AE5	2EA22D30	0F46317D	61B76178	E7BC1BB342393A06	4497F473D8AE5B3A	238B885A51663B54	FCFD9F88948D42A7
	317CF942	AA617101	B1B646B0	9FB8201C	5B6E332077A59C5D	C798AA981789AC8D	F916664458B5AD3F	F7086A16B2407A56
	31E77E87	0E87682D	AB27674A	1C00EF33	8DD6CEC45AC62D09	2C217A7DC1AB282C	8AA14855B8A7A065	1BA096650A8E8F6D
	49676DA0	5E36BB3F	369CB43A	F6E575E8	9ECAB9E7A91D59FE	A57F363A65CF10D3	F16FCED7A605DFE9	C02D0A46B23E8C31
6	472112C6	EBBA21DD	69FAF1B0	06AADA3C	2FCA68C9B1691627	59E2B79D4B2A88F8	D44A3CC624C9028F	6295CCEC81F0F5AF
	958968BA	FAF43AF0	8A346D6C	04DAD629	AFBA11EEC8CE43A4	A6BC58426BDAB6AC	C9FA0754D15A38A6	61B7C093B862D551
	28C63C70	F49BAA13	57DE5F7C	28841E18	B7A8A66A9227EE06	17BEF1A5F98B7250	CCAA13033F5ADCD3	15CBCEF3A8A993B5
	EA3F594F	8D744A62	57B54FF1	753A4160	2E321403DA39690B	D805E663071507B0	6D7EBAA185FF9F07	64071C2C7A0205EA
7	865ACF57	0B1CD341	44571AAD	1E351C75	BF643FF50F9B521B	D6ECDEF9B9AC18B0	29C44312EB0ED72A	6AA97E4B4BF39E0A
	679AB711	8D923CDC	115DC180	CF5E7435	A957D54C2B38DF1B	23E4928A7504F6B8	6CFEE0C2D418DC84	10464EB477E6D548
	94D66EB3	6B643DA7	C71FD3A8	EACD114A	18A96DABB8BBC145	406A6EE1C806F1E4	A54BD0A7B7291B4A	27BC2F8593DD77BE
	FE5A4582	101A0A61	DEF929CE	F81307CE	3BE8FF6116D7AFB0	4D78AEB59B3A9C25	9F03C664A44601DC	DDBE9B34DA020E59
8	EE830EF5	EFEDB52C	D9B5DDE0	11699703	F51507DD9E95189F	AB5E0B1641FAD08F	09B7BF70943B60DE	E35D03636672DACD
	A59F827F	E7DA769E	9ACF9688	FE6B4EE6	1D013C731A134DCD	850FC95D9CA677C8	48D78D3658CBE8D0	3898A93514FBF49D
	2D99EFFF	C1F42728	1B33FCE4	2484C32D	8849E2B60F59D433	A1C7E702A391D4B9	C0057990DE07D3EE	6BBF9A8B0E6CB108
	454DEF51	65220E90	D8B53023	10265221	7DE67998BA91A9CE	68F2B4BC4B8F6A52	4EFE2C5711E64647	27173B06EFB20807
9	5BA9F23B	4BF4491B	AEF87C06	ADCB6C25	002D7BC6826E5C4E	98EAB016A3E7207F	256F87371014F66D	6D9F12AF7B51BA9A
	84A8D85E	06C4583B	A845E529	E9B02D7F	DF7D5DFDED1E7078	E24E4E023117F906	D3BB9C3FE292CE3D	A67A8CE3BE764D9E
	EE6C0E9B	DEC3200A	6C54629E	B511AD99	45A7A7E1DBA2E0AA	5D54BBDD7E640E1D	088696EC342398E7	5542ABB8B94BB46B
	3D9F13B9	B8D1EDA8	864F0FFA	12F8AF0B	35DC03910937D57B	7A726243CEE6D7DA	8562A88AA0E1FF37	7B49C51D5DF11672
10	A61FD7E0	FF35B59A	F69A212D	47A15ADD	68B23861E57C9B05	F213CDD14E146238	0E2123C3E9EC08A3	9C6E6D6DAA93B9ED
	55D610B0	F029E3EB	0C00141B	DB2E13A3	A5702C464CC5B083	3A6519BEB2F56890	58F28DC1E45BF8DB	DEF3C026B70B8321
	6804E008	873E90C3	74385699	04D9596B	1E67BF80DAF92084	6CF77997E48644FE	C9B7FA02D70DB5AC	3FDC30C21AEE5282
	415BB241	BABB116D	9E823917	E2DE402B	3D6CBA696BA6DD93	244DBB6790CCB37D	D0E6766D04FAE591	2B8B936EEC644EF5
11	9041779F	33B0A06B	A3B26416	8A15CD3C	256DC2FF6DE0A8F9	C8283552487D4780	36E9CA389070DCA5	EFB083848F8A3405
	690B4EF7	31FAFE50	E588204D	7F2B5954	384CDB03F49E7149	7DE9B6194F547AAF	C492972CC2C2E3DE	32D560A0FFD100C7
	D42B9E7B	07A1E0D7	BAD18B76	6EDAC458	66EDB88FB0A8ED53	FC22F84560B7ABAF	9A63F2BF0970DC84	123A7129FF26A569
	5E73F7CA	B3EC7D7C	13822907	34427635	555B49259E70FD2B	A79F97997F5A58B5	1399D5D5EC0BD97A	110FC0B750934E61
12	E706F4FD	F789B2AD	0B0FA1F9	46D2ADFD	79970BBA797F440D	B0459551EFD2E5A0	E140D289FE653F5A	C8FC4FD63D554F1A
	A838E222	E5E99CE6	99AC76DD	19D0101A	37E8BBE4772E3497	2927C042216E82B4	BE2D1C1DA47865A1	5A50DCF8AE92DF6B
	73F58246	C6C1CB88	05985586	FED36478	8A7CB9A29451499B	2FC455A75270E7D3	A5D94AC009584F4B	2B6A365256A121AF
	0CB8A89B	560C0D0B	96B6CBFF	738DBBBE	8268A2C7ACA0B4CB	FB6D0AADC42F2F25	3D853A4E3B481849	F123FE58267C3EF7
13	26A1AE25	C23C5F1A	97569C36	45630F28	1B448E8F92662327	3591F3F3E586BB58	D6886C54B54AB965	4AD24A7E75C843E0
	F7A7003A	3E3F1431	5BFE2226	9C1B8620	907F731206F383F4	B7C30BE4C9CB308C	4FEC526992D7DF72	9D095158BAA00FED
	96FB64B1	DA975336	FD3DC3EE	EFAC8EED	15E38D27D3CBEA58	5AC8ODB9250ED949	131066D58338BA8D	921424D98DFB5ED2
	D0A74E35	4CCE7985	A73D02BC	7E819839	B0E50C530C665E5F	3B8D7BBCD6CFE748	0F91216102B8C6B9	DF3FBE9627CF0CB7
14	0EB1BCE0	315AFC74	DBAECA2E	7451F4BF	7DD0542A032CC09D	CD428DF51AD1B8FC	FFF2139B160E45B6	56CE164150489EB3
	ACAB0D51	E8385576	86F03E6B	016AE48C	0264062B41CE6E01	FEA2C239875FA55F	9D8BBA9828A7F43B	B1C12516C2FB1DE8
	E231BB88	DB29BED0	EC2622DE	7F6F0F74	4943B0FC28BB7A36	E54B13E241E5CC89	882C0750BDA3EB7C	97028319155939DE
	10A3EE22	7F5E0DA3	5AACEB4B	3B2FB855	441CD99CD0C48940	D3B65696FF5B41E9	5A0E09FA7BCCF7BC	41523B4C1E564872
15	340B3FA8	8B4EE0F6	EB62C2D7	9AF0EC8E	F2DC521CEBDE2E3D	C0921B301794BED5	7BCBEFD874ABE56B	35872E5C29899E44
	431C83CB	79E0F0D6	D83AEA5E	18D2DA56	FDE750453067080B	AEFE0DEA33D8EE2A	8E1741A7DEC189D0	897240121D56DCD3
	A008685C	CB836339	F6ED4128	7A9FC592	52CA19E5C5DD54BF	F0589BCF1B050432	52AED8532DE4DD9E	C11492B2FC68D82E
	1F7B1AD6	0E1DC227	3EB20808	C12CE387	0D4EA16E832DA7F3	AE57C79E053DF9C2	CC9358DDA6A3320C	D45F711CE4BDDE7C

Full AEAD Computations

Let K, N, A, P, B, C and T denote, key, nonce, header, payload, trailer, ciphertext and authentication tag. Unless stated otherwise, intermediate values are snapshots of the state after the final permutation of a given phase. For example, for $\mathsf{d} = 1$ the *end of*

the header processing denotes the state after final permutation F^r in the header phase or in other words, it is the state before the first payload data block is absorbed. This corresponds to the state after the fourth application of F^r in Figure 32, assuming that two header blocks are processed. We use the following notation for the particular phases:

[S]	basic state setup
[I]	end of initialisation
[A]	end of header processing
[P]	end of payload processing (for $D = 1$)
[P_i]	end of payload processing on lane L_i (for $D \neq 1$)
[M]	end of merging phase (for $D \neq 1$)
[B]	end of trailer processing

Values for NORX32

We assume that the following input data is given:

K	00112233	44556677	8899AABB	CCDDEEFF
N	FFFFFFFF	FFFFFFFF		
A	10000002	30000004		
P	80000007	60000005	40000003	20000001
B	null			

Padding of header and payload results in:

$\mathsf{pad}_{320}(A)$	10000002	30000004	00000001	00000000
	00000000	00000000	00000000	00000000
	00000000	80000000		
$\mathsf{pad}_{320}(P)$	80000007	60000005	40000003	20000001
	00000001	00000000	00000000	00000000
	00000000	80000000		
[S]	243F6A88	FFFFFFFF	FFFFFFFF	85A308D3
	00112233	44556677	8899AABB	CCDDEEFF
	13198A2E	03707344	254F537A	38531D48
	839C6E83	F97A3AE5	8C91D88C	11EAFB59

Note that the basic state [S] is the same for all of the following instances.

NORX32-4-1

[I]	A6859693	625D4B18	E04194D3	F5CB03FA
	AF507C8F	5D030D50	7D21E730	3801A9CB
	07B5E1D1	D04182D6	EBAB5473	D9D2769B
	08CEF45E	3C958F7B	EC346524	9F8DF11E

[A]	4CABE77F	D475C97A	144B3BC1	26DA08D5
	B1346EF8	A28DF65D	AC420E4C	7DD01B39
	4AA9652A	0604CC4A	B7B65DDE	EB10AAEE
	985FF6DF	76BE155F	B21C0D0D	E6F69429
[P]	F41C98A9	E9BEC3FE	80558E88	29A994CE
	BA66A2F0	BFA93172	A76EEBF6	96D5723C
	83F70E69	55004610	CDB24FF9	E9B4AFE6
	C5C78FE6	F6E9419C	8874C258	47E562F0
C	CCABE778	B475C97F	544B3BC2	06DA08D4
T	F41C98A9	E9BEC3FE	80558E88	29A994CE

NORX32-6-1

[I]	4CEC66FF	157BAAA2	6991EFFF	AADA81C1
	7247E0A2	80D3A609	93DC4EFD	807F171D
	6C103BC4	4EE90BAD	F9B77D28	2FDBBBA4
	C3727837	03558E04	965C1F7F	5199A103
[A]	FA1411D0	AB9C308F	7D955C15	312D5B92
	3A2632BD	E8A59A23	DD63C2CD	3C55DABC
	570FF833	9334EB2C	91CEC145	07E05F31
	B9F6732A	8135214C	28C91E39	345DE658
[P]	D949093E	7C630B9A	136DF6BA	4552A2D6
	4C414F3B	550670A3	13ED83CA	8D401751
	7ABFC5F1	8CC7C5AB	9224D2C4	F2D1079C
	64686B2E	4667414D	A325B4EE	18D98082
C	7A1411D7	CB9C308A	3D955C16	112D5B93
T	D949093E	7C630B9A	136DF6BA	4552A2D6

Values for NORX64

We assume that the following input data is given:

K	0011223344556677	8899AABBCCDDEEFF	FFEEDDCCBBAA9988	7766554433221100
N	FFFFFFFFFFFFFFFF	FFFFFFFFFFFFFFFF		
A	1000000000000002	3000000000000004		
P	8000000000000007	6000000000000005	4000000000000003	2000000000000001
B	null			

Padding of header and payload results in:

$\mathsf{pad}_{640}(A)$	1000000000000002	3000000000000004	0000000000000001	0000000000000000
	0000000000000000	0000000000000000	0000000000000000	0000000000000000
	0000000000000000	8000000000000000		
$\mathsf{pad}_{640}(P)$	8000000000000007	6000000000000005	4000000000000003	2000000000000001
	0000000000000001	0000000000000000	0000000000000000	0000000000000000
	0000000000000000	8000000000000000		

[S]	243F6A8885A308D3	FFFFFFFFFFFFFFFF	FFFFFFFFFFFFFFFF	13198A2E03707344
	0011223344556677	8899AABBCCDDEEFF	FFEEDDCCBBAA9988	7766554433221100
	A4093822299F31D0	082EFA98EC4E6C89	AE8858DC339325A1	670A134EE52D7FA6
	C4316D80CD967541	D21DFBF8B630B762	375A18D261E7F892	343D1F187D92285B

Note that the basic state [S] is the same for all of the following instances.

NORX64-4-1

[I]	F89FD073FCCF0DA6	AE43878AA7E72AF9	EF066EAD35656108	08C32CF2CF1906D0
	D2559C7E3422F141	E265687406901D3F	E3886E55AEF44CF5	05E29A496C6754CC
	2428DFC030ECEA24	EBFD8386ED3F66CF	8AC1A7A45D8359A9	4841831786961693
	6E00340067525020	D0501BCA0863DA3F	8BE6640FB8EDC928	9A3C3CEDC09CAC96

[A]	F0261529CACFB7EA	CDEDDCDEF81912B0	CD5DB2E73CB9A44D	776DD64D38BE869E
	B3B79BA058476D47	0EA348964F56ECB7	63978BC4A812319B	F01A4F387FBDAD4B
	23D339F601AFEA47	CD05C917848FB9B6	11D94C1706162801	67F3B5E71E0CF175
	94D78A64E58A1802	F2D722FC3A3DB407	715C705209A81DFE	A1B9C89A6C2ED8B7

[P]	97F45179DE5D5804	E47E0A8FA7B157D0	AD4E6B119FE2FEF2	939490F32AADB1B9
	E3BF4D22690AAC45	0E2BFC492EC8D7BB	33674AAA404FCD28	82BE08E20723D809
	ACEB91F64F0FDDF6	79CAADC699955BD6	7153D1074C4F27C5	08219F499EA851E3
	99CBB4749F0C30C1	5179D3688B3568F3	63BCF31B0ACC1379	07F0BE20069786E7

C	70261529CACFB7ED	ADEDDCDEF81912B5	8D5DB2E73CB9A44E	576DD64D38BE869F
T	97F45179DE5D5804	E47E0A8FA7B157D0	AD4E6B119FE2FEF2	939490F32AADB1B9

NORX64-6-1

[I]	D5CF5CD48C6868AB	0FAFBD1AA8039919	14F59C62FC9CB969	20F35D82D19578F6
	D592F5909E505B5D	BAD17EF8DC4DA18F	2C9F6B9D1D1DB330	B7970C4627D4C402
	72FACE8E660AF7B5	04861620AD8F57F6	983826D969E8D782	8F0B3F0ADF675061
	441841BEBC628A9B	FA207A4618DA0433	C06234917729C5AA	026A2D8A04CA2FBC

[A]	FF4A769F211E6B66	D985FF698E0D4527	CF8111E2A08AA8F0	A87CDFD1B311DE80
	7DDCDB2A8B824A07	84E2BA34D9612809	9BA303444722A6DE	0779B80D013AC008
	91B500ACA2D9F329	9DAA3C4A8BDE7EB8	E46216761A1A8521	A027691BC55EFA56
	DF30ECC2F1CE4718	B0BF2B6CD13BF396	5EFCD174822FEA78	7E56B9F928AAA1EF
[P]	F1ED4CE4283D7886	845717865C1CBC57	EF140434D843AAD5	5EA1E5958DA5E0A2
	03127D12277E4DAA	880F2B5ECFF1F81C	CE757C680DFD6FFF	FC52C828DF8F3F34
	90266271C89B608C	59849772DFB714B1	9BD7F7473F786C95	518E228E15785FA4
	AD1B9FB9EABD6C28	251E204596861386	3003E97AA52DCE02	B8E17372831F9161
C	7F4A769F211E6B61	B985FF698E0D4522	8F8111E2A08AA8F3	887CDFD1B311DE81
T	F1ED4CE4283D7886	845717865C1CBC57	EF140434D843AAD5	5EA1E5958DA5E0A2

NORX64-4-4

Due to the small size of the payload, only lane L_0 is processing actual payload data. Lanes L_1, L_2 and L_3 are processing only padding blocks.

[I]	338C058AE8194393	74778588AD8A0424	717B083B9CA3384A	F79F58E95EFF0E2C
	D987F5143B757F46	94CA10DE03307B7B	5A8C688836BE49A1	0C2615300BAB15D9
	2003EC1D9FA29CD2	3E8E1C08EFE04E51	9F58E4D329027F28	AFC1091A6701261A
	A87EB269C7E3B277	6F97C2B2D9E06578	B5C196FC526812EE	5215BC53F0A3704F
[A]	A20BA07081CA6E94	7FCAA889F5AD928B	BF180FA878FCF254	04A2798652A27E25
	425E2ABB4CB2515F	1F7ABC792FE0B467	97225637D513A823	B20441661B25249B
	6CE14936B9FAB518	A7C5DFFFABEBF53F	45B9FFCD0489E536	B3FAD2BD2EF458C8
	FA3498A1DB765EC7	4F2518E2F2FEFE40	0085565E3458FB72	A6177F6306FFCAF4
[P_0]	640E50FFE4DFCB45	5FD63593E2C1F15C	3A436E855A9A7E33	DE58B928F710239A
	8AB9C0DF7794890E	109E4CB5AC9D1EEE	CCC8F51A860E24E4	D9D2414EB38D1104
	36A40B1E6F829C5E	EFBDD5F21DDB3F12	7BA93218BE79BCFA	6360A7651D46740A
	F3F8F6948FD46921	2167B350D45F33FC	E0A250275D090CAC	83486343F67D0365
[P_1]	99E731AF490C669F	5E042CEF52716526	3C36EAB55D40A8EB	FBB1A9C1F8800232
	689A8358BB6E617D	1A0BD8AF1544CAB4	7C45D749A46C391F	069880E79310B20F
	520ECC298FCB0790	2A4DED2EA7DF4A11	97FE9C35E1136F61	9062F1BA7E7238AD
	945691C3E90A9860	FD2834091E07DB2F	EC2F097F358B502D	D7B6DB402AB3C679
[P_2]	13D0B4F055264930	FA93B1110AEC6F9D	66DD4249E0392DE1	2EBEC2E83AA88E41
	5EF53EE8D7C23A33	B6D38B6B66AF9861	80DFE51738BCCE72	C458876F558CB3F4
	3D349B0E50444ABF	E6C9596063E165FA	AF02CE82D6FD48EE	F7A59A2BDCD8968E
	B14BE3A0E7948DCF	558DD4B85F62E190	72813332E94E7875	73A587DE8DF25971

$[\mathsf{P}_3]$	60CC5AA0870C1E17	E31CF699B76B7E4F	EEF643013FEE4617	9063E2CAB510B49F
	68F7F0096920DF8A	AAA50B4FA5F3F8D1	49D574E292D9303A	F7B17A35F1958285
	D2985B123CD7369F	3CC6FC797310B62F	A4CFC266FFAE0213	146EE258880C1936
	C6A05A15367DE773	48FDCB711E06BC7F	A2235F75C78621C7	27FCCCB105D3B6F6
$[\mathsf{M}]$	365018CE729D30D7	D3FD630542509BB8	553581D9089B8641	943A4D6925C3BFD4
	2AA4790764766228	9776518212852CDB	15C15E54F23CCE12	6ABFF1840275DBD7
	B24AD552AD264B27	8CCEB9FC62B4F043	5273B139690DB1BB	B4C210991536C42C
	EDA68A52E2CA4793	CEBF29CBD161D3A5	71F32B3E343910D9	2F49A8CC450445A4
C	640E50FFE4DFCB45	5FD63593E2C1F15C	3A436E855A9A7E33	DE58B928F710239A
T	D6AE9B5944D4125C	78DD05B6721D277B	97C28DB804F05910	3C99D7CFE2B15138

Publications

Conference Papers

C. Beierle, P. Jovanovic, M. M. Lauridsen, G. Leander, and C. Rechberger. Analyzing Permutations for AES-like Ciphers: Understanding ShiftRows. *Topics in Cryptology — CT-RSA 2015*. K. Nyberg, editor. Volume **9048**, Lecture Notes in Computer Science. Springer, 2015, pp. 37–58.

P. Jovanovic and S. Neves. Practical Cryptanalysis of the Open Smart Grid Protocol. *Fast Software Encryption — FSE 2015*. Springer, 2015. (to appear)

P. Jovanovic and I. Polian. Fault-based Attacks on the Bel-T Block Cipher Family. *Design, Automation and Test in Europe — DATE 2015*. EDA Consortium, 2015, pp. 601–604.

P. Jovanovic, A. Luykx, and B. Mennink. Beyond $2^{c/2}$ Security in Sponge-Based Authenticated Encryption Modes. *Advances in Cryptology — ASIACRYPT 2014*. T. Iwata and P. Sarkar, editors. Volume **8873**, Lecture Notes in Computer Science. Springer, 2014, pp. 85–104.

R. Kumar, P. Jovanovic, W. Burleson, and I. Polian. Parametric Trojans for Fault-Injection Attacks on Cryptographic Hardware. *Fault Diagnosis and Tolerance in Cryptography — FDTC 2014*. IEEE, 2014.

J.-P. Aumasson, P. Jovanovic, and S. Neves. Analysis of NORX: Investigating Differential and Rotational Properties. *Progress in Cryptology — LATINCRYPT 2014*. D. F. Aranha and A. Menezes, editors. Volume **8895**, Lecture Notes in Computer Science. Springer, 2014, pp. 306–324.

J.-P. Aumasson, P. Jovanovic, and S. Neves. NORX: Parallel and Scalable AEAD. *European Symposium on Research in Computer Security — ESORICS 2014*. M. Kutylowski and J. Vaidya, editors. Volume **8713**, Lecture Notes in Computer Science. Springer, 2014, pp. 19–36.

R. Kumar, P. Jovanovic, and I. Polian. Precise Fault-Injections using Voltage and Temperature Manipulation for Differential Cryptanalysis. *20th International On-Line Testing Symposium — IOLTS 2014*. IEEE, 2014.

P. Jovanovic, M. Kreuzer, and I. Polian. A Fault Attack on the LED Block Cipher. *International Workshop on Constructive Side-Channel Analysis and Secure Design — COSADE 2012*. W. Schindler and S. Huss, editors. Volume **7275**, Lecture Notes in Computer Science. Springer, 2012, pp. 120–134.

Journal Papers

P. Jovanovic and M. Kreuzer. Algebraic Attacks using SAT-Solvers. *Groups — Complexity — Cryptology*, **2**, 2010.

Preprint Papers

J.-P. Aumasson, P. Jovanovic, and S. Neves. NORX8 and NORX16: Authenticated Encryption for Low-End Systems. Trustworthy Manufacturing and Utilization of Secure Devices — TRUDEVICE 2015.

P. Jovanovic, M. Kreuzer, and I. Polian. Multi-Stage Fault Attacks on Block Ciphers. 14th Workshop on RTL and High Level Testing — WRTLT 2013.

P. Jovanovic, M. Kreuzer, and I. Polian. An Algebraic Fault Attack on the LED Block Cipher. Third International Conference on Symbolic Computation and Cryptography — SCC 2012.